ALMA MATER STUDIORUM — UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in
Ingegneria e Scienze Informatiche

Multi-sensing Data Fusion: Target Tracking via Distributed Particle Filtering

Elaborato in
Sistemi Distribuiti

Relatore
Andrea Omicini

Co-Relatore
Giovanni Ciatto

Presentato da:
Alessandro Contro

Seconda Sessione di Laurea
Anno Accademico 2017-2018

KEYWORDS

Data Fusion
Multi Agent System
Particle Filtering
Jason
AgentSpeak
Target Tracking
Simulation

Abstract

Multi-sensing Data Fusion: Target Tracking via Distributed Particle Filtering

by Alessandro Contro

In this Master's thesis, Multi-sensing Data Fusion is firstly introduced with a focus on perception and the concepts that are the base of this work, like the mathematical tools that make it possible. Particle filters are one class of these tools that allow a computer to perform fusion of numerical information that is perceived from real environment by sensors. For this reason they are described and state of the art mathematical formulas and algorithms for particle filtering are also presented. At the core of this project, a simple piece of software has been developed in order to test these tools in practice. More specifically, a Target Tracking Simulator software is presented where a virtual trackable object can freely move in a 2-dimensional simulated environment and distributed sensor agents, dispersed in the same environment, should be able to perceive the object through a state-dependent measurement affected by additive Gaussian noise. Each sensor employs particle filtering along with communication with other neighboring sensors in order to update the perceived state of the object and track it as it moves in the environment. The combination of Java and AgentSpeak languages is used as a platform for the development of this application.

Contents

Chapter 1

Introduction

In the current days the interest in distributed autonomous systems research is at its peak. Just by looking at any engineering publication, it is easy to find articles talking about *Internet of Things* and *autonomous self driving cars*. These systems hugely rely on environmental awareness through the use of sensor arrays or sensor networks in order to perform their operations. This leads us to focus our interest on the mathematical and engineering processes that are at the base of perception carried out by a computer.

In this thesis we take on a journey to explore how this process of environmental awareness is performed by a computing machine, which brings us to introduce *Multi-Sensing Data Fusion* as the starting point for understanding how the environment is modeled inside a sensor-enabled agent. Considering that these agents can only "see" the environment at discrete times, the *Dynamic World Modeling framework* iteratively allows them to keep track of a set of environmental states, which form the agent's internal description of the world.

The world modeling iterative process can be done for numerical values perceived by the agent's sensors, and it is enabled by statistical methods such as *particle filters*. For this reason, after a brief introduction to distributed particle filtering, we try to master a specific technique to obtain a Particle Filter (PF) on an agent in a Multi-Agent System (MAS), which is the *Likelihood Consensus Distributed Particle Filter* (LCDPF) algorithm in a sensor-agent network.

To try out this new acquired knowledge we then try to implement a small MAS which simulates a sensor network tracking a moving object in a 2-dimensional virtual environment. For the implementation we make the architectural choice of using the Jason library in Java, which allows us to easily create and run logic for agents written in AgentSpeak language. For this reason we dedicate a section to introduce Jason and the AgentSpeak language, focusing on the features that are more relevant to this project.

The thesis is organized as follows: chapter 2 firstly gives an overview on the state of the for data fusion and distributed particle filtering. It then focuses more into LCDPF as it is the core topic of this thesis. In the same chapter a small introduction to Jason and AgentSpeak language is exposed. In chapter 3 the case study is introduced and with that the theory for particle filtering is given a context. Chapter 4 describes the architecture of the software application that we implement, and shows how the particle filtering techniques are translated into runnable Java and AgentSpeak code.

Chapter 2

State of the art

This chapter is mainly dedicated to introduce all the knowledge found in literature that has been used in the work for this project. Section 2.1 briefly describes AgentSpeak and the Jason library and focuses mainly on the features that are used in software developed for the case study later discussed in this work. Section 2.2 gives and overview on the foundations of data fusion which is *Dynamic World Modeling*. Finally section 2.3 firstly introduces the reader to particle filtering and shows various distributed approaches to this techniques, then it dives deep into the technique we decided to implement in this work, giving all the necessary mathematical foundations on which the software we developed is based on. Note that most of the information is directly quoted from the respective documents cited along the chapter.

2.1 AgentSpeak and Jason

On the contrary of traditional programs with a simple input-compute-output structure, agents and multi-agent systems are that kind of software that needs to maintain a long-term, ongoing interaction with their environment. An agent is a reactive system that exhibits some degree of autonomy in the sense that we delegate some task to it, and the system itself determines how best to achieve this task [4].

An agent is a system *situated* in a certain *environment* and by that we mean that the agent is capable of *sensing* their environment and that it is also capable of *acting* on that same environment via *effectors* and *actuators* in order to modify its state. Apart from being situated, an agent also shows other properties such as *autonomy*, *proactiveness*, *reactivity* and *social ability* [4].

It is more likely that these agents live in a multi-agent system where other agents are present and each one may have different characteristics, such as different behaviors. Due to their autonomous feature, agents may be created equal, but the evolution of the environment may lead them to act differently from each other. Each agent has a certain "sphere of influence" on the environment which most of the times overlaps with the sphere of influence of other agents, making the overall evolution of the system unpredictable.

2.1.1 The BDI Agent Model

BDI stands for *belief*, *desire* and *intention* and this kind of architecture originated from the theory of human practical reasoning. This model depicts the agent as if it had a mental state and the three elements of the model can be described as follows:

Beliefs are information that the agent has about the world, these information can come as perception, as knowledge from other agents or as knowledge inferred by the same agent (self).

Desires are all the possible states of affairs that the agent might like to accomplish. They are potential influencers of the agent's actions and it's reasonable to have conflicting desires.

Intentions are the states of affairs that the agent has decided to work towards. They may be goals that are delegated to the agent or may result form considering desires. Intentions are desires that the agent has *committed* to.

Agent-oriented Programming (AOP) offers a familiar and non-technical way to talk about complex systems, and it relieves the user from dealing with the control issues of traditional declarative programming, in other words the user tells what the system should achieve and the built-in control mechanism figures out how to reach that goal.

2.1.2 Agent Communications

So far what we discussed above deals with the internal operations of a single agent, but as we said it's more likely that an agent operates in a system with other agents and that requires social interactions via a certain communication architecture. Communication in multi-agent systems is usually based on the *speech-act theory* according to which that "language is action", in the sense that an agent attempts to change the world with the use of communication as much as the use of actual actions. The difference between *speech actions* and *non-speech actions* is that the domain of the former is limited to the mental state(s) of the recipient of the act, while the latter directly interact with the environment.

Speech acts, also referred as *performatives*, are classified according to their *illocutionary force* which can be listed as follows:

- *representatives*, passing simple information
- *directives*, which attempt to get the recipient to do something
- *commissives*, which state the commitment of the speaker
- *expressives*, whereby a speaker expresses a mental state
- *declaration*, such as declaring a state

For the purpose of communication among agents a dedicated language has been developed and it's called Knowledge Query and Manipulation Language (KQML). It is a dedicated high-level communication language which defines a number of performatives such as *tell*, which is at the base of agent communication in Jason. Based on KQML a standard was developed which is called FIPA (Foundation for Intelligent Physical Agents). This standard aims to simplify and rationalise the performative set as much as possible, and to address the issue of semantics. This should impose certain rules on the way communication is perfomed among agents via KQML which in the end should allow for a certain level of interoperability between BDI agents developed with different languages or between agents living on different systems.

2.1.3 Jason Agent Programming Language

According to the BDI architecture the main components of a Jason agent are the *belief base*, which can be updated accordingly after perceiving the environment, and the agent's *goals* which are achieved by the execution of *plans*. BDI agents, and so Jason agents, are reactive planning systems which permanently run and react to some form of *events*. The reaction to these events is carried on with said plans.

The interpretation of the agent program determines the agent's reasoning cycle. The agent is constantly perceiving the environment, reasoning about how to act so as to achieve its goals, then acting so as to change the environment. The reasoning is done according to the plans that the agent has in its *plan library*. When the agent is created at runtime, the plan library consists of the plans that the programmer writes as an AgentSpeak program.

Next a brief description of the main components of an AgentSpeak/Jason agent:

Beliefs An agent has a belief base, which in its simplest form is a collection of literals, as in traditional logic programming where information is represented in symbolic form by *predicates*. Predicates in the belief base can be accompanied by *annotations* which are Prolog structures that provide details that are strongly associated with one particular belief. Annotation can contain any kind of information, but only specific kind of annotation have a meaning for the Jason interpreter. One of those is the *source* annotation which contains the source of the associated belief. The main kinds of source are *perceptual information*, *communication* from other agents and *mental notes* generated by the same agent (self notes).

```
//sensor 1 knows its name
name(sensor1)[source(self)]
//sensor 1 has been told that sensor2 is its neighbor
//by sensor2 itself
neighbor(sensor2)[source(sensor2)]
```

Goals Goals express the properties of the states of the world that the agent wishes to bring about. When representing a goal in a agent program, it means the agent is committed to act so as to change the world to a state in which the agent will believe that the goal is true, by sensing the environment. AgentSpeak offers *achievement* goals and *test* goals. The former is meant as we described above, where the agent commits to act and change the world until a certain belief becomes true. The latter is a way to retrieve information from the belief base, although they can still lead to the execution of plans in certain circumstances.

```
//the agent has the goal to move left
!move(left)
//this will trigger a plan that should provide the agent
//with a percept telling it has successfully moved right
   or not
```

Plans They are the agent's know-how and they are usually triggered by changes in the agent's beliefs and goals. A plan is composed by three distinct parts: the *triggering event*, the *context* and the *body*. The triggering event tells the agent, for each of the plans in their plan library, which are the specific events for which the plan is to be used. Context is used for checking the current situation so as to determine whether a particular plan, among the various alternatives, is

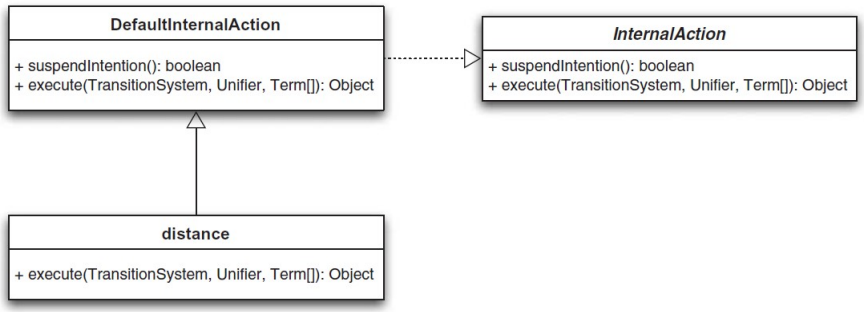

FIGURE 2.1: Example interface of a custom internal action (source [4]).

likely to succeed in handling the event, given the latest information the agent has about the environment. Then at last the body is a sequence of formulæ determining a course of action that will hopefully succeed in handling the event that triggered the plan.

```
triggering_event : context <- body.
```

Another important tool that an agent has is the *action*. Actions allow the agent to act within an environment and they represent what the agent is able to do. In other words they are a symbolic representation of "hardware" actions. Executing an action on the environment doesn't give a direct feedback on its success, so it is necessary that the agent has a way to perceive if something changed due to the execution of that action. As example while moving a robot arm, it has to give a feedback that the motion was completed or not, or if the arm performed some changes in the environment, the agent should be able to perceive them through its sensors.

While actions can be seen as something acting outside of the agent, there is a distinction with another kind of action, called *internal action*. This kind of formula allows to do some reasoning within the agent. Internal actions can be customized by programmers to extend the AgentSpeak language with operations written in Java that are not otherwise available (fig. 2.1).

Jason offers a number of standard internal actions, one of the most important for communication is the internal action *.send* which allows the exchange of messages between agents. The general form of the pre-defined internal action for communication is:

```
.send(receiver,illocutionary_force,propositional_content)
```

where the *receiver* is the agent name, or a list of names, given during configuration (if more instances of the same agent are present, Jason adds a number after the name starting by 1). The *illocutionary force* denotes the kind of message that the agent is sending, and the available options are:

- **tell**: *s* intends *r* to believe (that *s* believes) the literal in the message's content to be true;
- **untell**: *s* intends *r* not to believe (that *s* believes) the literal in the message's content to be true;
- **achieve**: *s* requests *r* to try and achieve a state of affairs where the literal in the message content is true (goal delegation);

- **unachieve**: *s* requests *r* o drop the goal of achieving a state of affairs where the message content is true;
- **askOne**: *s* wants to know if the content of the message is true for *r*;
- **askAll**: *s* wants all of *r*'s answers to a question;
- **tellHow**: *s* informs *r* of a plan (passing know-how);
- **untellHow**: *s* requests that *r* disregard a certain plan (delete plan from *r*'s plan library);
- **askHow**: *s* wants all of *r*'s plans that are relevant for the triggering event in the message content.

The *propositional content* is usually a literal and is the content of the message.

2.1.4 Simulated Environment in Jason

One of the key aspects of autonomous agents is that they are situated in an environment. It can be the real world or the Internet, while some other times it is necessary to create a computational model that is able to simulate the dynamic aspect of a real environment, e.g. for testing or for research. It is important that the model creates a certain level of transparency at the agent, so it can operate as if it was in the real world environment (figure 2.2).

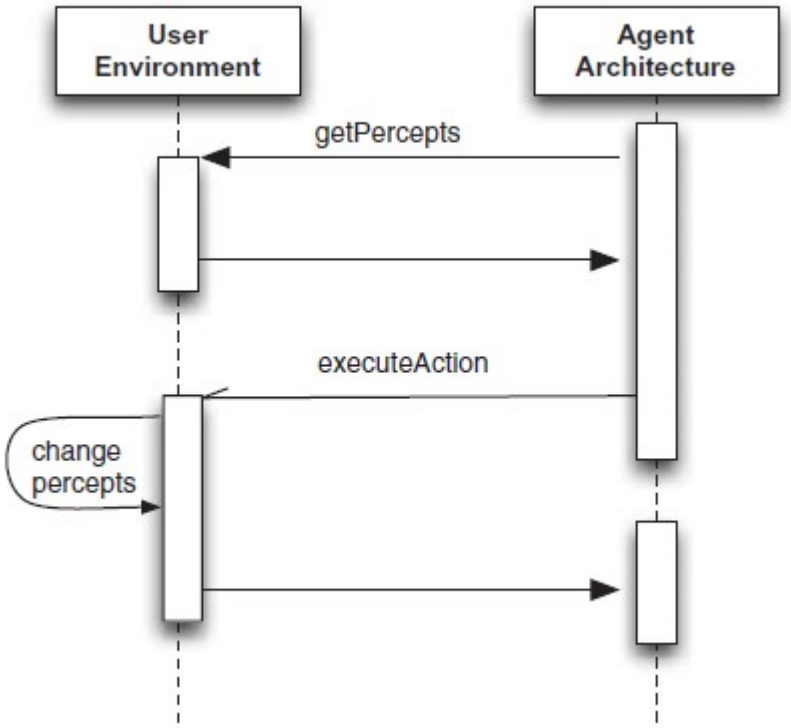

FIGURE 2.2: Interaction between agent and environment (source [4]).

Jason provides the *Environment* interface which can be extended by the user, it can be parametrized in the *init* method and logic for agents' actions can be defined

in the method *executeAction*, which is called by the underlying infrastructure whenever and agent performs an action (fig. 2.2). The whole class can then be customized like a regular Java class. It is possible for example to create an UI that draws relevant agent operations on screen. The interface finally offers methods that deal with agent perception, so percepts can be added or removed from an agent's belief base and the agent will always see these beliefs with the note *source(percept)*. Generally a

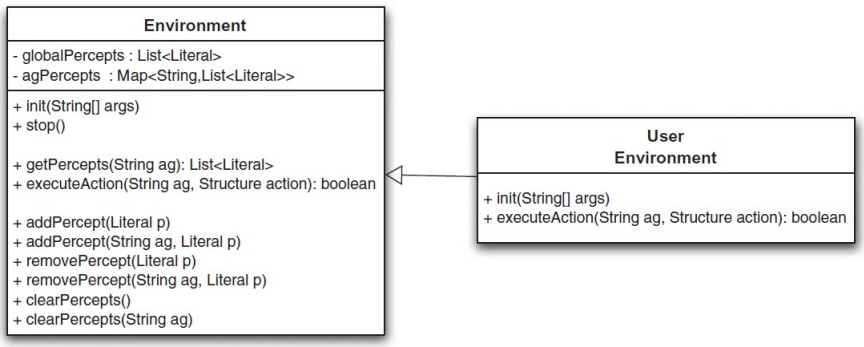

FIGURE 2.3: Environment interface and extendable methods in Jason (source [4]).

simulated environment in Jason is composed by the main class Environment which interacts with the agents and by other support classes such as a *model* class that tracks relevant information, and a *view* which renders the model on screen. In figure 2.3 it's possible to see all methods that Jason provides for the Environment class. It should be noted that the environment is able to access all agents' percepts and modify them woth the the field *agPercepts* and the methods *add/remove/clear Percept*, which can be targeted to all agents or a single one. Note also that the word *percept* refers only to beliefs that are generated by the environment, so other kind of beliefs that an agent has in their belief base should not be accessible from the outside.

2.2 Perception and Data Fusion

An agent, seen as something that "acts" in a certain environment, needs first of all to be aware of the state of that same environment and in order to do that it needs to create and maintain an internal coherent description of that state through the interpretation of different discrete observations performed by sensing means usually captured at different times. These interpretation can then be integrated and used to maintain the internal description of the world, or the part of it that is interesting to the agent. All this process of acquisition, interpretation and integration of observations into a coherent internal description of the world is at the base of *perception* and takes the name of *Dynamic World Modeling* [5], where "dynamic" means that the model changes with time, taking advantage of its relative continuity.

2.2.1 Perception

Perception is not a goal, but a mean for an agent to acquire information from the environment in order to execute actions towards a certain goal and for this the agent needs to have a description of that environment. Perception is defined as:

> The process of maintaining of an internal description of the external environment. [5]

Somebody could think that it would be easier to just use the actual environment as the description model but that would require an extremely complete and rapid sensing ability from the agent and most of the times the entirety of the world contains way much more information than the agent needs for its goals.

A generic framework for *Dynamic World Modeling* is presented in [5] which we are going to describe and it's pictured in figure 2.4. In the framework, independent observations are translated into a common coordinate space and vocabulary and then fused into a model by a cyclic 3-phases process. The phases are *Predict*, *Match* and *Update*.

During the prediction phase, the current state of the model is used to "guess" the state of the observed external world according to a transition model at the time when the next observation is taken. During the matching phase the observations are used to align the model to the external world. While during the update phase, the information obtained from the new observations is integrated into the model. This cycle can be used both to add new information to the model and to remove "old" or "incorrect" information from it, which also prevents the model from growing with no limits.

2.2.2 Principles for integrating perceptual information

The authors in [5] identify the following set of principles for data fusion.

> 1) Primitives in the world model should be expressed as a set of properties.

A primitive should express an association of properties that describes the state of some part of the world and is most likely paired with a value of precision. This is known as *state vector*.

> 2) Observation and Model should be expressed in a common coordinate system.

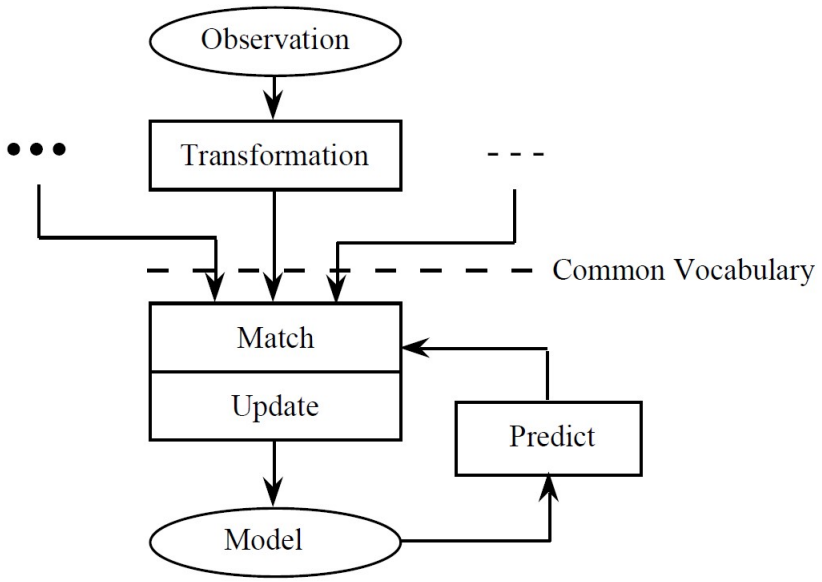

FIGURE 2.4: Dynamic World Modeling framework scheme.
Source [5]

When the model represents the state in a different way than the pure observation, a transformation function should be applied to the latter. This usually requires knowledge of the sensor geometry and function. The most used coordinate systems can be scene based or observer based. The decision of which system to use is related to the cost of the transformation and the use case.

3) Observation and model should be expressed in a common vocabulary.

A model can be seen as a state database, values referring to the same property should be labeled the same way. A good approach to integrate information coming from different (kinds of) sensors is to define a standard primitive that contains all the properties that can be observed or inferred by the different sensors. This way each sensor can supply a complementary subset of properties of the state vector.

4) Properties should include an explicit representation of uncertainty.

Precision and *Confidence*. The former can be seen as a spatial uncertainty, often represented by a probability function of noise. The latter indicates how close the agent thinks the property is to the the real value.

5) Primitives should be accompanied by a confidence factor.

Primitives should be considered as hypotheses that the agent makes on the external world. Each hypotheses should include a value of likelihood which can be expressed as a probability. This way information that is uncertain at first can be confirmed or deleted from the model after some iterations.

2.2.3 Techniques for Numerical Data Fusion

When dealing with primitives composed by numerical property estimates and their precision, a well defined set of techniques is available from Mathematical and Statistical studies. One powerful tool that is also the scope of this work is the *particle filter* which will better discussed later on.

A dynamic world model $M(t)$, is a list of primitives which describe the state of a part of the world at an instant in time t.

$$M(t) \equiv \{P_1(t), \ldots, P_m(t)\}$$

Each primitive $P_i(t)$ in the world model describes a local part of the world as a conjunction of estimated properties $\hat{X}(t)$. plus a unique *id* and a confidence factor $C(t)$

$$P(t) \equiv \{id, \hat{X}(t), C(t)\}$$

The confidence factor allows the system to filter the good primitives from the bad ones, keeping the former and removing the latter. Successive observations change the confidence value, increasing it the more the primitive is close to the value observed. Primitives that obtain a low confidence factor are considered as noise and removed from the model. High confidence on certain elements allows them to remain relevant for several cycles, even in the case the part of the world observed is obscured.

A primitive is an estimate of a part of the world as a set of N properties, represented by the vector $\hat{X}(t)$.

$$\hat{X}(t) \equiv \{\hat{x}_1(t), \ldots, \hat{x}_n\}$$

$X(t)$ is the actual state of the external world, and it is estimated byt the observation process $\mathbf{H}$ which projects the world onto an observation vector $Z(t)$, and it's corrupted by random noise $N(t)$

$$Z(t) \equiv \mathbf{H}\, X(t) + N(t)$$

The world state $X(t)$ is not directly knowable and the estimate $\hat{X}(t)$ should converge to that through successive observations. At each cycle the system produces an estimate $\hat{X}(t)$ by combining a predicted observation $\bar{Z}(t)$ with and actual observation $Z(t)$. The difference between the predicted vector $\bar{Z}(t)$ and the observed vector $Z(t)$ provides the basis for updating the estimate $\hat{X}(t)$.

Both the estimate and the observation must be accompanied by a value of uncertainty which provides the tolerance bounds when matching observation and predictions, and provides the relative strength of prediction and observation when calculating a new estimate [5].

2.3 Particle Filtering

Particle filtering is a Monte Carlo approximation of optimal sequential Bayesian state estimation. It has become the methodology of choice for nonlinear and non-Gaussian systems and due to the increasing interests in sensing agent networks a lot of research in Distributed Particle Filters (DPF) has been made. Because of the distributed nature of agent networks, the measurements are dispersed among the agents and for that diffusing the locally available information throughout the network becomes an essential component of DPFs.

Agents cooperatively estimate certain parameters of the surrounding environment based on their local measurements and they need to cooperate among each other because their local measurements are usually insufficient for obtaining reliable and complete estimates.

For distributed estimation algorithms, the communication aspects of the agent network are very important, such as the communication topology, usually described as a graph, and the properties of the links.

In these kind of systems it is assumed that the locations of the agents are known and that each single agent is estimating its position related to the environment. This information can either be self-acquired by the agent (in the case of mobile agents) or given as parameter to the agent at the start of its operation (in the case of fixed agents).

Additional aspects influencing the design of distributed estimation algorithms include energy constraints, such as limited battery, and computation constraints, such as limited transmission power and range and latency. The algorithms may also have to meet other requirements related to the application like robustness to link and node failure and scalability.

We will now provide a top-down overview of particle filters before focusing on a specific PF technique. Particularly on this chapter and consequently along the rest of this thesis, we heavily rely on Particle Filtering theory presented in [6], [7] and [8].

2.3.1 Sequential Bayesian Estimation

Distributed Particle Filtering is a variant of the centralized Sequential Bayesian Estimation, so before moving on could be useful to review these principles first.

Consider a time-dependent state vector $\mathbf{x}_n$, n being a discrete time index, that evolves according to the *system model*

$$\mathbf{x}_n = \mathbf{g}_n(\mathbf{x}_{n-1}, \mathbf{u}_n), \quad n = 1, 2, \ldots. \tag{2.1}$$

$\mathbf{g}_n(\cdot, \cdot)$ is a known (usually nonlinear) function and $\mathbf{u}_n$ is white driving noise that is independent of the past and present states and whose probability density function (pdf) is known. At time n, measurement vector $\mathbf{z_n}$ is observed, which relates to $\mathbf{x}_n$ via the *measurement model*

$$\mathbf{z_n} = \mathbf{h}_n(\mathbf{x}_n, \mathbf{v}_n), \quad n = 1, 2, \ldots. \tag{2.2}$$

$\mathbf{h}_n(\cdot, \cdot)$ is a known (usually nonlinear) function and $\mathbf{v}_n$ is white measurement noise that is independent of the past and present states and of the driving noise and whose pdf is known. $\mathbf{z}_{1:n} \triangleq (\mathbf{z}_1^T \ldots \mathbf{z}_n^T)^T$ is defined as the vector for all measurements up to time n. Equation 2.1 and 2.2 together with the statistical assumption determine a probabilistic formulation of the system model by the state-transition pdf $f(\mathbf{x}_n|\mathbf{x}_{n-1})$ and of the measurement model by the likelihood function $f(\mathbf{z}_n|\mathbf{x}_n)$, both

of which are allowed to be time-varying. From the statistical assumptions it also follows that $f(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{z}_{n-1}) = f(\mathbf{x}_n|\mathbf{x}_{n-1})$, which means that the state $\mathbf{x}_n$ is conditionally independent of all past measurements $\mathbf{z}_{1:n-1}$, and that $f(\mathbf{z}_n|\mathbf{x}_{n-1}, \mathbf{z}_{n-1}) = f(\mathbf{z}_n|\mathbf{x}_n)$, which means that $\mathbf{z}_n$ is conditionally independent of all past measurements $\mathbf{z}_{1:n-1}$, given the current state $\mathbf{x}_n$.

The task is the estimation of the state $\mathbf{x}_n$ from all measurements up to time n, $\mathbf{z}_{1:n}$, in a sequential (recursive) manner. The Bayesian approach is to calculate the posterior pdf $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ and with that one can then calculate various estimates of $\mathbf{x}_n$ such as the *minimum mean-square error* (MMSE) estimator of $\mathbf{x}_n$ obtained as the mean of $f(\mathbf{x}_n|\mathbf{z}_{1:n})$

$$\hat{\mathbf{x}}_n^{\text{MMSE}} \triangleq \mathrm{E}[\mathbf{x}_n|\mathbf{z}_{1:n}] = \int \mathbf{x}_n f(\mathbf{x}_n|\mathbf{z}_{1:n}) d\mathbf{x}_n, \quad n = 1, 2, \ldots. \tag{2.3}$$

It can be shown that the posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ can be calculated sequentially from the previous posterior $f(\mathbf{x}_{n-1}|\mathbf{z}_{1:n-1})$ and the measurement vector $\mathbf{z}_n$, in two steps. First, in the *prediction step*, the "predicted posterior" $f(\mathbf{x}_n|\mathbf{z}_{1:n-1})$ is calculated from the previous posterior $f(\mathbf{x}_{n-1}|\mathbf{z}_{1:n-1})$ and the state-transition pdf $f(\mathbf{x}_n|\mathbf{x}_{n-1})$ according to

$$f(\mathbf{x}_n|\mathbf{z}_{1:n-1}) = \int f(\mathbf{x}_n|\mathbf{x}_{n-1}) f(\mathbf{x}_{n-1}|\mathbf{z}_{1:n-1}) d\mathbf{x}_{n-1}, \quad n = 1, 2, \ldots. \tag{2.4}$$

In the *update step*, the predicted posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n-1})$ is converted to the posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ according to

$$f(\mathbf{x}_n|\mathbf{z}_{1:n}) = \frac{f(\mathbf{z}_n|\mathbf{x}_n) f(\mathbf{x}_n|\mathbf{z}_{1:n-1})}{f(\mathbf{z}_n|\mathbf{z}_{1:n-1})}, \quad n = 1, 2 \ldots. \tag{2.5}$$

Note that this involves the likelihood function $f(\mathbf{z}_n|\mathbf{x}_n)$. The recursion for $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ established by 2.4 and 2.5 is initialized by$f(\mathbf{x}_n|\mathbf{z}_{1:n})|_{n=0} = f(\mathbf{x}_0)$.

A straightforward calculation of relation 2.3-2.5 is usually infeasible, since an analytical solution is in most cases unavailable and a numerical implementation involves the computation of multidimensional integrals. An important exception is the special case of linear system and measurement models with Gaussian driving and measurement noises and a Gaussian prior $f(\mathbf{x}_0)$. Usually this class of problems is solved with Kalman Filters or other similar approaches like the Gaussian sum filter [6] which are suboptimal solutions that may lead to large errors or even divergence.

2.3.2 The Particle Filter

A Particle Filter performs a Monte Carlo simulation-based approximation of optimal sequential Bayesian estimation in (2.3)-(2.5). The non-Gaussian posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ is represented by a set $\{(\mathbf{x}_n^{(j)}, w_n^{(j)})\}_{j=1}^J$ of randomly drawn samples or *particles* $\mathbf{x}_n^{(j)}$ and corresponding weights $w_n^{(j)}$ which establishes a discrete approximation of the posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n}) \approx \sum_{j=1}^J w_n^{(j)} \delta(\mathbf{x}_n - \mathbf{x}_n^{(j)})$. ($\delta(\cdot)$ denotes the multidimensional Dirac delta function). With this representation, one can obtain various estimates of $\mathbf{x}_n$. In particular (2.3) can be approximated as

$$\hat{\mathbf{x}}_n^{\text{MMSE}} \approx \hat{\mathbf{x}}_n \triangleq \sum_{j=1}^{J} \mathbf{x}_n^{(j)} w_n^{(j)}. \tag{2.6}$$

This approximation is accurate if the number of particles $\mathbf{x}_n^{(j)}$ located in regions of significant probability mass is sufficiently large and if the weights $w_n^{(j)}$ are calculated appropriately.

At time n the Particle Filter recursively updates the previous particles $\mathbf{x}_{n-1}^{(j)}$ and weights $w_{n-1}^{(j)}$ using the observation $\mathbf{z}_n$. More specifically, the particle representation of the posterior is used to approximate the prediction step in (2.4) and the update step in (2.5). This is done by means of *importance sampling*, whereby the samples $\mathbf{x}_n^{(j)}$ are randomly drawn from a specified pdf that is different from the posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$. The prediction step (2.4) is performed by sampling from a proposal pdf $\mathbf{q}(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{z}_n)$, thus obtaining particles that are used to approximate the predicted posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n-1})$. The update step (2.5) is performed by computing the particle weights $w_n^{(j)}$ using the likelihood function $f(\mathbf{z}_n|\mathbf{x}_n)$.

Generic Particle Filter (SIR Filter) Algorithm

Here we present the *sequential importance sampling* (SIR) filter algorithm from where the particle filter in this project, as many other well-known particle filters, is derived.

At time $n = 0$, the algorithm is initialized by J particles $\mathbf{x}_0^{(j)}$, $j = 1, \ldots, J$ drawn from a prior pdf $f(\mathbf{x}_0)$. The weights of the particles are initially equal ($w_0^{(j)} = 1/J \quad \forall j$). At time $n \geq 1$, the following steps are performed:

1. *Prediction step*: For each previous particle $\mathbf{x}_{n-1}^{(j)}$, a new particle $\mathbf{x}_n^{(j)}$ is sampled from a suitably chosen proposal pdf $q(\mathbf{x}_n|\mathbf{x}_{n-1}^{(j)}, \mathbf{z}_n) \equiv q(\mathbf{x}_n|\mathbf{x}_{n-1}, \mathbf{z}_n)|_{\mathbf{x}_{n-1}=\mathbf{x}_{n-1}^{(j)}}$.

2. *Update step*: Non-normalized weights associated with the particles $\mathbf{x}_n^{(j)}$ drawn on the previous step are calculated according to

$$\tilde{w}_n^{(j)} = w_{n-1}^{(j)} \frac{f(\mathbf{z}_n|\mathbf{x}_n^{(j)}) f(\mathbf{x}_n^{(j)}|\mathbf{x}_{n-1}^{(j)})}{q(\mathbf{x}_n^{(j)}|\mathbf{x}_{n-1}^{(j)}, \mathbf{z}_n)}, \quad j = 1, \ldots, J. \tag{2.7}$$

   The weights are then normalized according to $w_n^{(j)} = \tilde{w}_n^{(j)} / \sum_{j'=1}^{J} \tilde{w}_n^{(j')}$. The set of particles and weights $\{(\mathbf{x}_n^{(j)}, w_n^{(j)})\}_{j=1}^{J}$ represents the posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$.

3. *Calculation of estimate*: From the set of particles and weights $\{(\mathbf{x}_n^{(j)}, w_n^{(j)})\}_{j=1}^{J}$, an approximation of the MMSE state estimate is computed according to (2.6) $\hat{\mathbf{x}}_n = \sum_{j=1}^{J} w_n^{(j)} \mathbf{x}_n^{(j)}$.

4. *Resampling*: The set $\{(\mathbf{x}_n^{(j)}, w_n^{(j)})\}_{j=1}^{J}$ can be resampled if necessary. The resampled particles are obtained by sampling with replacement from the set $\{(\mathbf{x}_n^{(j)}\}_{j=1}^{J}$, where $\mathbf{x}_n^{(j)}$ is sampled with probability $w_n^{(j)}$. This produces J resampled particles $\mathbf{x}_n^{(j)}$. The weights are redefined to be identical, i.e., $w_n^{(j)} = 1/J$.

Note that resampling is done to avoid the *degeneracy problem* [3], where, after a few iterations, all but one particle will have a negligible weight. The idea of resampling is to eliminate particles that have small weights so to concentrate on particles with large weights.

2.3.3 Distributed Sequential Bayesian Estimation

In a distributed Agent Network setting, the measurements are dispersed among the agents. Let us consider an AN consisting of K agents. At time n, agent $k \in \{1, \ldots, K\}$ observes a local measurement vector $\mathbf{z}_{n,k}$, which is relate to the state $\mathbf{x}_n$ via the *local measurement model* [cf. (2.2)]

$$\mathbf{z}_{n,k} = \mathbf{h}_{n,k}(\mathbf{x}_n, \mathbf{v}_{n,k}), \quad n = 1, 2, \ldots. \tag{2.8}$$

Here $\mathbf{h}_{n,k}(\cdot, \cdot)$ is a known, agent-dependent, generally nonlinear function and $\mathbf{v}_{n,k}$ is a local measurement noise that is white and independent of the past and present states and of the driving $\mathbf{u}_n$ in (2.1). The global (all-agents) measurement vector $\mathbf{z}_n$ in (2.2) is now given by the collection of all local measurement vectors in (2.8), i.e., $\mathbf{z}_n \triangleq (\mathbf{z}_{n,1}^T, \ldots, \mathbf{z}_{n,K}^T)^T$. Equation (2.8) together with the statistical assumptions determines the *local likelihood function* $f(\mathbf{z}_{n,k}|\mathbf{x}_n)$ of agent k. Bayesian estimation assumes that $\mathbf{v}_{n,k}$ is independent of the local measurement noises of the other agents, $\mathbf{v}_{n,k'}$ for $k' \neq k$. This implies that the *global likelihood function* $f(\mathbf{z}_n|\mathbf{x}_n)$ factorizes into the local likelihood functions, i.e.,

$$f(\mathbf{z}_n|\mathbf{x}_n) = \prod_{k=1}^{K} f(\mathbf{z}_{n,k}|\mathbf{x}_n). \tag{2.9}$$

The goal of distributed sequential estimation is to estimate $\mathbf{x}_n$ in a sequential manner, based on the measurements $\mathbf{z}_{n',k}$ of all (or relevant subset of) agents k for all times n' up to the current time n. Preferably, each agent should communicate only with the neighboring agents, and the estimation results should be available at each agent or at least at a relevant subset of agents.

2.3.4 Distributed Particle Filtering

To obtain a Distributed Particle Filter implementation, some approximations are typically required. Further approximations may be needed to reduce computation and communication requirements. As a result, DPFs usually do not perform as well as a centralized PF that has access to all the measurements locally. However, avoiding these approximations will usually imply excessive communication requirements, poor scalability, and practically inconvenient constraints such as the need for synchronized random generators.

The existing DPF algorithms differ in aspects such as type and amount of the data communicated between the agents, communication range, local processing, computational complexity, memory requirements, estimation accuracy, robustness, scalability, and latency. A basic distinction can be made between algorithms that employ a central processing unit, called fusion center (FC), and those that operate in a decentralized manner. Figure 2.5 shows the taxonomy for distributed particle filters. In the remainder of this section a short summary is given explaining briefly each category.

Based on the type of data communicated between the agents, we will discriminate two broad classes of decentralized DPFs: *statistics dissemination-based* DPFs, where processed data (representations of posteriors or likelihood functions) are exchanged between the agents, and *measurement dissemination-based* DPFs, where raw or quantized measurements are exchanged. A subdivision of the statistics dissemination-based class can be based on the set of agents that perform particle filtering at any given time, with corresponding differences regarding agent scheduling and communication topology. Accordingly, we will distinguish between *leader agent (LA)-based*

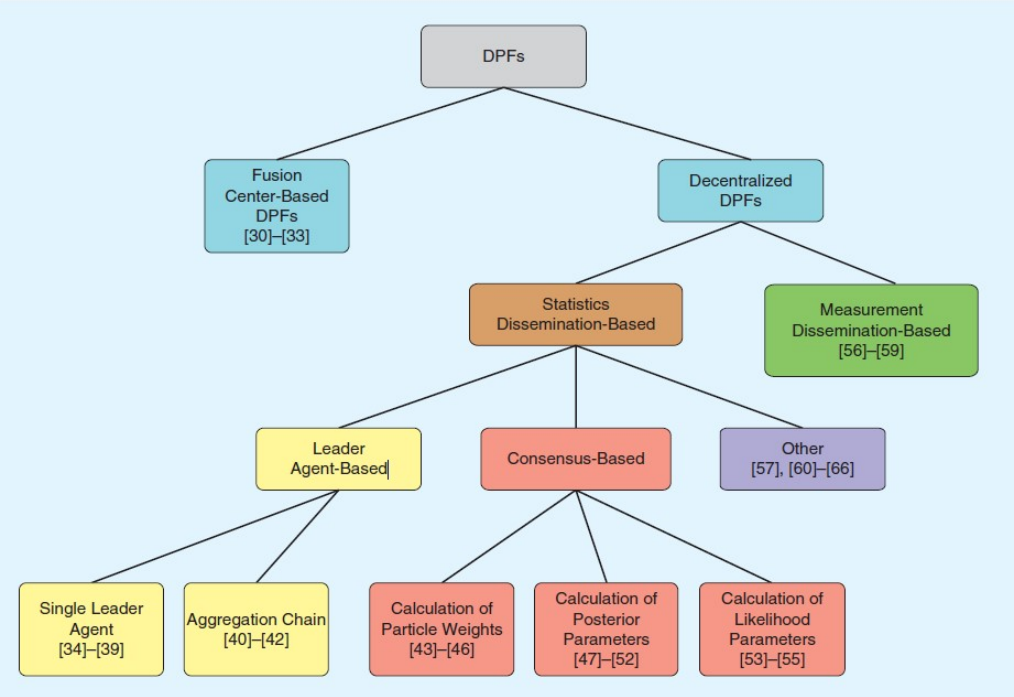

FIGURE 2.5: Taxonomy of DPFSs. (source [6])

DPFs, where at any given time only one agent is in charge of the processing, and *consensus-based* DPFs, where all the agents in the network process data simultaneously.

2.3.5 Consensus-Based DPFs

With consensus-based DPFs, all agents perform particle filtering simultaneously and possess a particle representation of a posterior. Ideally, this posterior is the global posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ reflecting the current and past measurements of all agents, $\mathbf{z}_{1:n}$. This goal is achieved, or, at least approached by a decentralized consensus algorithm that establishes an agreement on certain global quantities across all agents. These quantities are then used by each agent to establish a local approximate particle representation of the global posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$. The use of consensus algorithms implies that each agent transmits certain quantities to a set of neighboring agents.

The advantages of a consensus approach:

- consensus only requires communication with neighboring agents
- no need for routing protocols or global knowledge of the network
- each agent is in possession of the global estimate
- robustness to changes of then network topology, link failures and agent failures

while the main disadvantages:

- higher communication requirements compared to Leader Agent-based approach
- the number of consensus iterations to diffuse the information may increase with the size of the network.

Next we'll briefly describe the three classes of Consensus based particle filtering algorithms.

Consensus-based calculation of particle weights This class of DPF performs a distributed computation of global particle weights $w_n^{(j)}$ (reflecting the measurements of all agents) from local weights $w_{n,k}^{(j)}$ (each reflecting only the measurement of the respective agent k). For each weight $w_n^{(j)}, j \in \{1, \ldots, J\}$, one consensus algorithm for computing an average is executed. This approach presupposes that identical sets of particles $\{\mathbf{x}_n^{(j)}\}_{j=1}^J$ are sampled at each agent. This, in turn, requires that the local random number generators at the agents are synchronized, such that the same pseudo-random numbers are obtained in the entire network, and that the identical particle representations $\{(\mathbf{x}_{n-1}^{(j)}, w_{n-1}^{(j)})\}_{j=1}^J$ of the previous global posterior $f(\mathbf{x}_{n-1}|\mathbf{z}_{1:n-1})$ are available at each agent.

Consensus-based calculation of posterior parameters In DPFs with this approach, the global posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ is approximated by a Gaussian or Gaussian Mixture pdf. A similar parametric approximation is used for the local posteriors $f(\mathbf{x}_n|\mathbf{z}_{1:n-1}, \mathbf{z}_{n,k})$ incorporating the past measurements of all agents, $\mathbf{z}_{1:n-1}$, and the current local measurement of agent k, $\mathbf{z}_{n,k}$. The parameters of the global posterior are calculated from the parameters of the local posteriors in a distributed manner using consensus algorithms.

Consensus-based calculation of likelihood parameters This is a class of DPFs in which consensus algorithms are used to compute the *global likelihood function* (GLF) (sometimes also called *joint likelihood function* (JLF)) $f(\mathbf{z}_n|\mathbf{x}_n)$ rather than the posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$. The GLF, or at least an approximation, is obtained at each agent as a function of $\mathbf{x}_n$. This allows each agent to evaluate the GLF at the particles $\mathbf{x}_n^{(j)}$, which is required for calculating the weights $w_n^{(j)}$ in the particle filter update step (2.7). Therefore each agent is able to locally run a PF that is equivalent to a global PF because it uses the GLF and calculates a global estimate involving the all-agents measurement vector $\mathbf{z}_n$. The local PFs operate independently of each other (only the GLF is computed cooperatively), so that a synchronization of random number generators is not required. Note that global estimates obtained at individual agents may differ slightly. More in depth detail are give in the dedicated section (2.4)

Consensus algorithms

Consensus algorithms are effective tools for diffusing information in distributed computations. In the context of agent networks, "consensus" means a global agreement on some quantity that depends on the data of all agents, and a "consensus algorithm" specifies the corresponding information exchange between neighboring agents and the computations performed by each agent. Consensus algorithms are iterative schemes that diffuse information through the network and, usually, reach a global agreement only asymptotically. The differences between the values at the individual agents after a finite number of iterations depends on the number of iterations, the size and topology of the network, and the particular consensus algorithm. Consensus algorithms are advantageous in that there is no bottleneck or single point of failure, and the algorithms are robust to changing network topology and unreliable network conditions such as link failures.

The most common consensus algorithms used in DPFs are the *average consensus*, the *randomized gossip*, and the *max-consensus*. Then general situation for when a consensus algorithm can be used, is when each agent $k \in \{1, \dots, K\}$ possesses a scalar quantity s_k and the result of the disseminated information can be calculated with a sum, like for example the average $\frac{1}{K}\sum_{k=1}^{K} s_k$ or a mathematical function such as $\max_{k\in\{1,\dots,K\}}\{s_k\}$, so the result is available at each agent. At each iteration i of the consensus algorithm each agent updates an internal state using the internal states of a set of neighboring agents. At each agent the following steps are performed in the consensus process:

1. Initialize the internal state.
2. For $i = 1, \dots, I$ (I is the max number of iterations), update the internal state with a function $u(\cdot)$ which combines the previous internal state at $i-1$ with the previous internal states of all neighboring agents $k' \in \mathcal{N}_k$. The new internal state is broadcast to all neighbors $k' \in \mathcal{N}_k$.

The function $u(\cdot)$ depends on the type of consensus algorithm:

- *Average consensus* The new internal state is a linear combination of the previous internal state and the previous internal states of the neighbors. Each element of the combination has a weight defined by a dedicated function. For $I \to \infty$, each internal state converges to the average.
- *Randomized consensus* The new internal state is the average between the previous internal state and the previous internal state of an agent randomly picked from the set of neighbors of agent k. Again for $I \to \infty$, each internal state converges to the average.
- *Max-consensus* The new internal states becomes the max among all the internal states. With the appropriate changes it is possible to find the min as well. For $I \to \infty$, each internal state converges instead to the max/min.

2.4 Likelihood Consensus Distributed Particle Filtering

In sections (2.3.1) and (2.3.3) we discussed how Sequential Bayesian Estimation works, how it operates in a distributed context and how it dictates the core behavior of the particle filter algorithm. As we said in (2.3.5) *likelihood consensus distributed particle filtering* (LCDPF) means that each sensor in a network runs a local particle filter and all the sensors agree on the same, or almost the same, approximation of likelihood function via a consensus algorithm, so we now discuss how this approximation takes place.

2.4.1 Approximation of the Joint Likelihood Function

The likelihood consensus method can always be used it the local likelihood functions belong to the exponential family of distributions [7]. This requires an approximation of the local likelihood functions, and consequently of the JLF.

We assume that the local likelihood functions (seen as conditional pdf of $\mathbf{z}_{n,k}$) belongs to the exponential family of distributions and it can be written as

$$f(\mathbf{z}_{n,k}|\mathbf{x}_n) = c_{n,k}(\mathbf{z}_{n,k}) \exp(\mathbf{a}_{n,k}^T(\mathbf{x}_n)\mathbf{b}_{n,k}(\mathbf{z}_{n,k}) - d_{n,k}(\mathbf{x}_n)), \quad k = 1, \dots, K, \tag{2.10}$$

with some time- and sensor-dependent functions $c_{n,k}(\cdot) \in \mathbb{R}_+$, $\mathbf{a}_{n,k}(\cdot) \in \mathbb{R}^q$, $\mathbf{b}_{n,k}(\cdot) \in \mathbb{R}^q$, and $d_{n,k}(\cdot) \in \mathbb{R}_+$, with arbitrary $q \in \mathbb{N}$. It is furthermore assumed that sensor k knows its own functions $c_{n,k}(\cdot)$, $\mathbf{a}_{n,k}(\cdot)$, $\mathbf{b}_{n,k}(\cdot)$ and $d_{n,k}(\cdot)$, but not $c_{n,k'}(\cdot)$, $\mathbf{a}_{n,k'}(\cdot)$, $\mathbf{b}_{n,k'}(\cdot)$ and $d_{n,k'}(\cdot)$ for $k' \neq k$. Using the factorization in (2.9), the JLF is obtained as

$$f(\mathbf{z}_n|\mathbf{x}_n) = \prod_{k=1}^{K} c_{n,k}(\mathbf{z}_{n,k}) \exp(\mathbf{a}_{n,k}^T(\mathbf{x}_n)\mathbf{b}_{n,k}(\mathbf{z}_{n,k}) - d_{n,k}(\mathbf{x}_n)) \tag{2.11}$$

$$= C_n(\mathbf{z}_n) \exp(S_n(\mathbf{z}_n, \mathbf{x}_n)), \tag{2.12}$$

where

$$C_n(\mathbf{z}_n) \triangleq \prod_{k=1}^{K} c_{n,k}(\mathbf{z}_{n,k}) \tag{2.13}$$

and

$$S_n(\mathbf{z}_n, \mathbf{x}_n) \triangleq \sum_{k=1}^{K} [\mathbf{a}_{n,k}^T(\mathbf{x}_n)\mathbf{b}_{n,k}(\mathbf{z}_{n,k}) - d_{n,k}(\mathbf{x}_n)]. \tag{2.14}$$

Note that the JLF (viewed as the conditional pdf of $\mathbf{z}_n$) also belongs to the exponential family. Thus, according to (2.12), for global inference based on all-sensors measurement vector $\mathbf{z}_n$, each sensor needs to know $S_n(\mathbf{z}_n, \mathbf{x}_n)$ as a function of $\mathbf{x}_n$, for the observed (fixed) $\mathbf{z}_n$. However, calculation of $S_n(\mathbf{z}_n, \mathbf{x}_n)$ at a given sensor according to (2.14) presupposes that the sensor knows the measurement $\mathbf{z}_{n,k}$ and the functions $\mathbf{a}_{n,k}(\cdot)$, $\mathbf{b}_{n,k}(\cdot)$ and $d_{n,k}(\cdot)$ for *all* sensors, i.e., for all k. Transmitting the necessary information from each sensor to each other sensor may be infeasible [7], [8].

As we discussed in (2.3.5) a powerful approach to diffusing local information through a sensor network is given by consensus algorithms. Unfortunately, a consensus-based distributed calculation of $S_n(\mathbf{z}_n, \mathbf{x}_n)$ is not possible in general because the terms of the sum in (2.14) depend on the unknown state $\mathbf{x}_n$ [7]. Therefore, an approximation of $S_n(\mathbf{z}_n, \mathbf{x}_n)$ is used that involves a set of coefficients not dependent on $\mathbf{x}_n$. The approximation is induced by the following approximations of the functions $\mathbf{a}_{n,k}(\cdot)$ and $d_{n,k}(\cdot)$ in terms of given basis functions $\{\varphi_{n,r}(\mathbf{x}_n)\}_{r=1}^{R_a}$ and $\{\psi_{n,r}(\mathbf{x}_n)\}_{r=1}^{R_d}$, respectively:

$$\mathbf{a}_{n,k}(\mathbf{x}_n) \approx \tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n) \triangleq \sum_{r=1}^{R_a} \alpha_{n,k,r} \varphi_{n,r}(\mathbf{x}_n) \tag{2.15}$$

$$d_{n,k}(\mathbf{x}_n) \approx \tilde{d}_{n,k}(\mathbf{x}_n) \triangleq \sum_{r=1}^{R_d} \gamma_{n,k,r} \psi_{n,r}(\mathbf{x}_n). \tag{2.16}$$

Here, $\alpha_{n,k,r} \in \mathbb{R}^q$ and $\gamma_{n,k,r} \in \mathbb{R}$ are expansion coefficients that do not depend on $\mathbf{x}_n$. $\alpha_{n,k,r}$ are referred to as coefficients even though they are vector-valued. The basis functions $\varphi_{n,r}(\mathbf{x}_n)$ and $\psi_{n,r}(\mathbf{x}_n)$ do not depend on k, which means the same basis functions are used by all sensors. They are allowed to depend on n although time-independent basis functions may often be sufficient [7]. It is assumed that sensor k knows the basis functions $\varphi_{n,r}(\mathbf{x}_n)$ and $\psi_{n,r}(\mathbf{x}_n)$, as well as the coefficients $\alpha_{n,k,r}$ and $\gamma_{n,k,r}$ corresponding to its own functions $\mathbf{a}_{n,k}(\mathbf{x}_n)$ and $d_{n,k}(\mathbf{x}_n)$, respectively; however it does not know the coefficients of other sensors, $\alpha_{n,k',r}$ and $\gamma_{n,k',r}$ with $k' \neq k$. The coefficient $\alpha_{n,k,r}$ and $\gamma_{n,k,r}$ can either be precomputed, or each sensor can compute them locally.

Substituting $\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n)$ for $\mathbf{a}_{n,k}(\mathbf{x}_n)$ and $d_{n,k}(\mathbf{x}_n)$ for $\tilde{d}_{n,k}(\mathbf{x}_n)$ in (2.14), the following approximation of $S_n(\mathbf{z}_n, \mathbf{x}_n)$ is obtained:

$$\begin{aligned}\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n) &\triangleq \sum_{k=1}^{K} \left[\tilde{\mathbf{a}}_{n,k}^T(\mathbf{x}_n) \mathbf{b}_{n,k}(\mathbf{z}_{n,k}) - \tilde{d}_{n,k}(\mathbf{x}_n) \right] \\ &= \sum_{k=1}^{K} \left[\left(\sum_{r=1}^{R_a} \alpha_{n,k,r}^T \varphi_{n,r}(\mathbf{x}_n) \right) \mathbf{b}_{n,k}(\mathbf{z}_{n,k}) - \sum_{r=1}^{R_d} \gamma_{n,k,r} \psi_{n,r}(\mathbf{x}_n) \right].\end{aligned} \tag{2.17}$$

By changing the order of summation, it is further obtained

$$\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n) = \sum_{r=1}^{R_a} A_{n,r}(\mathbf{z}_n) \varphi_{n,r}(\mathbf{x}_n) - \sum_{r=1}^{R_d} \Gamma_{n,r} \psi_{n,r}(\mathbf{x}_n), \tag{2.18}$$

with

$$A_{n,r}(\mathbf{z}_n) \triangleq \sum_{k=1}^{K} \alpha_{n,k,r}^T \varphi_{n,r}(\mathbf{x}_n), \quad \Gamma_{n,r} \triangleq \sum_{k=1}^{K} \gamma_{n,k,r}. \tag{2.19}$$

Finally, substituting $\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n)$ from (2.18) for $S_n(\mathbf{z}_n, \mathbf{x}_n)$ in (2.12), an approximation of the JLF is obtained as

$$\tilde{f}(\mathbf{z}_n|\mathbf{x}_n) \propto \exp(\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n)) = \exp\left(\sum_{r=1}^{R_a} A_{n,r}(\mathbf{z}_n) \varphi_{n,r}(\mathbf{x}_n) - \sum_{r=1}^{R_d} \Gamma_{n,r} \psi_{n,r}(\mathbf{x}_n) \right) \tag{2.20}$$

This shows that a sensor that knows $A_{n,r}(\mathbf{z}_n)$ and $\Gamma_{n,r}$ can evaluate an approximation of the JLF (up to a $\mathbf{z}_n$-dependent but $\mathbf{x}_n$-independent normalization factor) for all values of $\mathbf{x}_n$. The vector of all coefficients $A_{n,r}(\mathbf{z}_n)$ and $\Gamma_{n,r}$, $\tilde{\mathbf{t}}_n(\mathbf{z}_n) \triangleq (A_{n,1}(\mathbf{z}_n \cdots A_{n,R_a}(\mathbf{z}_n)\ \Gamma_{n,1} \cdots \Gamma_{n,R_d})^T$, can be viewed as a *sufficient statistic* that epitomizes the total measurement $\mathbf{z}_n$ within the limits of the approximation [7]. Because of (2.20), this sufficient statistic fully describes the approximate JLF $\tilde{f}(\mathbf{z}_n|\mathbf{x}_n)$ as a function of $\mathbf{x}_n$.

The expressions (2.18) and (2.19) allow a distributed calculation of $\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n)$ and, in turn, of $\tilde{f}(\mathbf{z}_n|\mathbf{x}_n)$ by means of consensus algorithms, due to the following key facts:

(i) The coefficient $A_{n,r}(\mathbf{z}_n)$ and $\Gamma_{n,r}$ do not depend on the state $\mathbf{x}_n$ but contain the information of all sensors (sensor measurement $\mathbf{z}_{n,k}$ and the approximations coefficients $\alpha_{n,k,r}$ and $\gamma_{n,k,r}$)

(ii) The state $\mathbf{x}_n$ enters into $\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n)$ only via the functions $\varphi_{n,r}(\cdot)$ and $\psi_{n,r}(\cdot)$, which are sensor independent and known to each sensor

(iii) According to (2.19), the coefficients $A_{n,r}(\mathbf{z}_n)$ and $\Gamma_{n,r}$ are sums in which each term contains only local information of a single sensor.

These facts form the basis of the LC method [7], [8].

Normalization factor If this factor is required at each sensor, it can also be computed via consensus algorithm if we consider the expression (2.13) and apply the logarithm

$$\log C_n(\mathbf{z}_n) = \sum_{k=1}^{K} \log c_{n,k}(\mathbf{z}_{n,k}). \tag{2.21}$$

Since this is a sum and $c_{n,k}(\mathbf{z}_{n,k})$ is known at each sensor, a consensus algorithm can be used for a distributed calculation of $\log C_n(\mathbf{z}_n)$

Basis Functions and Coefficients Approximation

$\varphi_{n,r}(\cdot)$ and $\psi_{n,r}(\cdot)$ are basis functions which can be monomials, orthogonal polynomials, and Fourier basis functions. The choice of the functions affects the accuracy, computational complexity, and communication requirements of the LC method.

Polynomial approximation A way that the expansion (2.15) can be approximated is through a polynom:

$$\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n) = \sum_{\mathbf{r}=\mathbf{0}}^{R_p} \alpha_{n,k,\mathbf{r}} \prod_{m=1}^{M} x_{n,m}^{r_m} \tag{2.22}$$

where $\mathbf{r} \triangleq (r_1 \cdots r_M) \in \{0, \ldots, R_p\}^M$; R_p is the degree of the multivariate vector-valued polynomial $\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n)$; $\sum_{\mathbf{r}=\mathbf{0}}^{R_p}$ is short for $\sum_{r_1=0}^{R_p} \cdots \sum_{r_M=0}^{R_p}$ with the constant $\sum_{m=1}^{M} r_m \leq R_p$; and $\alpha_{n,k,\mathbf{r}} \in \mathbb{R}^q$ is the coefficient vector associated with the basis function (monomial) $\varphi_{n,\mathbf{r}}(\mathbf{x}_n) = \prod_{m=1}^{M} x_{n,m}^{r_m}$ (here, $x_{n,m}$ denotes the m-th entry of $\mathbf{x}_n$). (2.22) in its expanded form looks like:

$$\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n) = \sum_{r_1=0}^{R_p} \cdots \sum_{r_M=0}^{R_p} \alpha_{n,k,r_1,\ldots,r_M} \, x_{n,1}^{r_1} \cdots x_{n,M}^{r_M} \tag{2.23}$$

(2.22) can be written in form of (2.15) by a suitable index mapping $(r_1 \ldots r_M) \in \{0, \ldots, R_p\}^M \leftrightarrow r \in \{1, \ldots, R_a\}$, where $R_a = \binom{R_p+M}{R_p}$. An analogous polynomial expansion can be used for $\tilde{d}_{n,k}(\mathbf{x}_n)$ in (2.16).

A convenient method for calculating the approximations of $\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n)$ and $\tilde{d}_{n,k}(\mathbf{x}_n)$ is given by a form of regression called Least Squares (LS) Fitting in which the J predicted particles at sensor k are used in data pairs $\{(\mathbf{x}_{n,k}^{(j)}, \mathbf{a}_{n,k}(\mathbf{x}_{n,k}^{(j)}))\}_{j=1}^{J}$ to approximate the $\alpha_{n,k,r}$ coefficients that describe the polynom for $\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n)$ so that $\sum_{j=1}^{J} \| \tilde{\mathbf{a}}_{n,k}(\mathbf{x}_{n,k}^{(j)}) - \mathbf{a}_{n,k}(\mathbf{x}_{n,k}^{(j)}) \|^2$ is minimized.

2.4.2 Sequential LC Algorithm

Once we are able to describe the full likelihood function via basis expansion approximation and we computed the local expansion coefficient to form a local sufficient statistic, we are ready to proceed with the consensus algorithm to obtain the global sufficient statistic that describes the JLF. A *linear* consensus is used, although other consensus algorithms can be used.

The *Sequential Likelihood Consensus* (SLC) Algorithm is an evolution of the *Dynamic Consensus* Algorithm presented in [8]. The main difference is that SLC is able to update the set of coefficients in a *single* step per each time n, based only on its previous value and the current measurements, reducing the overall latency and the transmitted data volume among sensors. In what follows $\mathcal{N}_k \subseteq \{1, \ldots, K\} \setminus \{k\}$ denotes a fixed set of sensors that are neighbors of sensor k. The operation performed by a fixed sensor k are explained, which are performed by all sensors simultaneously. The SLC algorithm can be summarized as follows.

Sequential Likelihood Consensus (SLC) Algorithm [8] At each time step $n \geq 1$, each sensor k performs the following tasks (note that the steps for the two sets of coefficients are the same, beside the use of the respective values for R and the initialization of the internal state) :

1. The coefficients $\{\alpha_{n,k,r}\}_{r=1}^{R_a}$ and $\{\gamma_{n,k,r}\}_{r=1}^{R_d}$ of the approximations are calculated via regression.

2. *Dynamic consensus algorithm* (single step) - $A_{n,r}(\mathbf{z_n})$ for each $r \in \{1,\dots,R_a\}$:

   (a) If $n = 1$, the internal state of sensor k is initialized as $s_{0,k,r} = \alpha_{1,k,r}^T \mathbf{b}_{1,k}(\mathbf{z}_{1,k})$

   (b) A temporary internal state is computed

   $$\zeta_{n,k,r} = \mu s_{n-1,k,r} + (1-\mu)\alpha_{n,k,r}^T \mathbf{b}_{n,k}(\mathbf{z}_{n,k}) \tag{2.24}$$

   where $\mu \in [0,1]$ is a tuning parameter.

   (c) The temporary internal state $\zeta_{n,k,r}$ is broadcast to all neighbors $k' \in \mathcal{N}_k$.

   (d) The internal state $s_{n,k,r}$ is calculated from the local and neighboring temporary internal states, $\zeta_{n,k,r}$ and $\{\zeta_{n,k',r}\}_{k'\in\mathcal{N}_k}$ by a single consensus step:

   $$s_{n,k,r} = W_{k,k}\zeta_{n,k,r} + \sum_{k'\in\mathcal{N}_k} W_{k,k'}\zeta_{n,k',r} \tag{2.25}$$

   The weights $W_{k,k'}$ are Metropolis weights:

   $$W_{k,k'} = \begin{cases} \frac{1}{1+\max(|\mathcal{N}_k|,|\mathcal{N}_{k'}|)}, & k' \neq k, \\ 1 - \sum_{k''\in\mathcal{N}_k} W_{k,k''} & k' = k \end{cases} \tag{2.26}$$

   (e) The internal state $s_{n,k,r}$ is scaled as $\tilde{A}_{n,r}(\mathbf{z_n}) \triangleq K s_{n,k,r}$

3. *Dynamic consensus algorithm* (single step) - $\Gamma_{n,r}(\mathbf{z_n})$ for each $r \in \{1,\dots,R_d\}$: repeat the same steps as for $A_{n,r}(\mathbf{z_n})$ but initialize the internal state at $n = 1$ and compute the temporary state with $\gamma_{1,k,r}$ and $\gamma_{n,k,r}$ respectively. While $\tilde{\Gamma}_{n,r}(\mathbf{z_n}) \triangleq K s_{n,k,r}$

4. By substituting $\tilde{A}_{n,r}(\mathbf{z_n})$ for $A_{n,r}(\mathbf{z_n})$ and $\tilde{\Gamma}_{n,r}(\mathbf{z_n})$ for $\Gamma_{n,r}(\mathbf{z_n})$ in (2.20), sensor k is able to obtain a consensus approximation of the JLF $\tilde{f}(\mathbf{z}_n|\mathbf{x}_n)$ for any given value of $\mathbf{x}_n$.

Note that with this algorithm the total number of sensors K present in the network has to be known at each sensor.

2.4.3 JLF Approximation with Gaussian Measurement Noise

In this section the polynomial approximation introduced in (2.4.1) is applied to the special case of (generally nonlinear) measurement functions and independent additive Gaussian measurement noise at the various sensors.

Measurement Model

The dependence of the sensor measurement $\mathbf{z}_{n,k}$ on the state $\mathbf{x}_n$ is described by the local likelihood functions $f(\mathbf{z}_{n,k}|\mathbf{x}_n)$. Let's assume that the measurements are modeled as

$$\mathbf{z}_{n,k} = \mathbf{h}_{n,k}(\mathbf{x}_n) + \mathbf{v}_{n,k}. \quad k = 1,\dots,K, \tag{2.27}$$

where $\mathbf{h}_{n,k}(\cdot)$ is the *measurement function* of sensor k and $\mathbf{v}_{n,k}$ $\mathcal{N}(\mathbf{0},\mathbf{Q}_{n,k})$ is zero-mean Gaussian measurement noise that is independent of $\mathbf{x}_{n'}$ for all n'. It is furthermore assumed that $\mathbf{v}_{n,k}$ and $\mathbf{v}_{n',k'}$ are independent unless $(n,k) = (n',k')$. Under these

assumptions, the $\mathbf{z}_{n,k}$ are conditionally independent given $\mathbf{x}_n$ and (2.9) holds. The local likelihood function of sensor k is given by

$$f(\mathbf{z}_{n,k}|\mathbf{x}_n) = \bar{c}_{n,k}\exp\left(-\frac{1}{2}[\mathbf{z}_{n,k} - \mathbf{h}_{n,k}(\mathbf{x}_n)]^T\mathbf{Q}_{n,k}^{-1}[\mathbf{z}_{n,k} - \mathbf{h}_{n,k}(\mathbf{x}_n)]\right), \tag{2.28}$$

with $\bar{c}_{n,k} \triangleq [(2\pi)^{N_{n,k}}\det\{\mathbf{Q}_{n,k}\}]^{-1/2}$. Then using (2.9), the JLF obtained is

$$f(\mathbf{z}_n|\mathbf{x}_n) = \bar{c}_n\exp\left(-\frac{1}{2}\sum_{k=1}^{K}[\mathbf{z}_{n,k} - \mathbf{h}_{n,k}(\mathbf{x}_n)]^T\mathbf{Q}_{n,k}^{-1}[\mathbf{z}_{n,k} - \mathbf{h}_{n,k}(\mathbf{x}_n)]\right), \tag{2.29}$$

with $\bar{c}_n = \prod_{k=1}^{K}\bar{c}_{n,k}$.

The local likelihood function $f(\mathbf{z}_{n,k}|\mathbf{x}_n)$ in (2.28) is a special case of the exponential family (2.10), with

$$\begin{aligned}
\mathbf{a}_{n,k}(\mathbf{x}_n) &= \mathbf{h}_{n,k}(\mathbf{x}_n),\\
\mathbf{b}_{n,k}(\mathbf{z}_{n,k}) &= \mathbf{Q}_{n,k}^{-1}\mathbf{z}_{n,k},\\
c_{n,k}(\mathbf{z}_{n,k}) &= \bar{c}_{n,k}\exp\left(-\frac{1}{2}\mathbf{z}_{n,k}^T\mathbf{Q}_{n,k}^{-1}\mathbf{z}_{n,k}\right),\\
d_{n,k}(\mathbf{x}_n) &= \frac{1}{2}\mathbf{h}_{n,k}^T(\mathbf{x}_n)\mathbf{Q}_{n,k}^{-1}\mathbf{h}_{n,k}(\mathbf{x}_n).
\end{aligned} \tag{2.30}$$

Consequently,

$$S_n(\mathbf{z}_{n,k}, \mathbf{x}_n) = \sum_{k=1}^{K}\mathbf{h}_{n,k}^T(\mathbf{x}_n)\mathbf{Q}_{n,k}^{-1}\left[\mathbf{z}_{n,k} - \frac{1}{2}\mathbf{h}_{n,k}(\mathbf{x}_n)\right]. \tag{2.31}$$

It is now possible to approximate $\mathbf{a}_{n,k}(\mathbf{x}_n)$ and $d_{n,k}(\mathbf{x}_n)$ by truncated basis expansion $\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n)$ and $\tilde{d}_{n,k}(\mathbf{x}_n)$. According to (2.30), approximating $\mathbf{a}_{n,k}(\mathbf{x}_n)$ is equivalent to approximating the sensor measurement function $\mathbf{h}_{n,k}(\mathbf{x}_n)$. Thus,

$$\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n) = \tilde{\mathbf{h}}_{n,k}(\mathbf{x}_n) = \sum_{r=1}^{R_a}\alpha_{n,k,r}\varphi_{n,r}(\mathbf{x}_n). \tag{2.32}$$

Furthermore, an approximation of $d_{n,k}(\mathbf{x}_n)$ can be obtained in an indirect way by substituting in (2.30) the above approximation $\tilde{\mathbf{h}}_{n,k}(\mathbf{x}_n)$ for $\mathbf{h}_{n,k}(\mathbf{x}_n)$; this yields

$$\begin{aligned}
\tilde{d}_{n,k}(\mathbf{x}_n) &= \frac{1}{2}\tilde{\mathbf{h}}_{n,k}^T(\mathbf{x}_n)\mathbf{Q}_{n,k}^{-1}\tilde{\mathbf{h}}_{n,k}(\mathbf{x}_n)\\
&= \frac{1}{2}\sum_{r_1=1}^{R_a}\sum_{r_2=1}^{R_a}\alpha_{n,k,r_1}^T\mathbf{Q}_{n,k}^{-1}\alpha_{n,k,r_2}\varphi_{n,r_1}(\mathbf{x}_n)\varphi_{n,r_2}(\mathbf{x}_n).
\end{aligned} \tag{2.33}$$

Using a suitable index mapping $(r_1, r_2) \in \{1_m \ldots, R_a\} \times \{1, \ldots, R_a\} \leftrightarrow r \in \{1, \ldots, R_a\}$, it is possible to write 2.33 in the form

$$\tilde{d}_{n,k}(\mathbf{x}_n) = \sum_{r=1}^{R_d}\gamma_{n,k,r}\psi_{n,r}(\mathbf{x}_n), \tag{2.34}$$

with $R_d = R_a^2$, $\gamma_{n,k,r} = \frac{1}{2}\alpha_{n,k,r_1}^T\mathbf{Q}_{n,k}^{-1}\alpha_{n,k,r_2}$, and $\psi_{n,r}(\mathbf{x}_n) = \varphi_{n,r_1}(\mathbf{x}_n)\varphi_{n,r_2}(\mathbf{x}_n)$ It is easily

verified that with this special basis expansion approximation of $d_{n,k}(\mathbf{x}_n)$, the resulting approximate JLF can be written as

$$\tilde{f}(\mathbf{z}_n|\mathbf{x}_n) = \bar{c}_n \exp\left(-\frac{1}{2}\sum_{k=1}^{K}[\mathbf{z}_{n,k} - \tilde{\mathbf{h}}_{n,k}(\mathbf{x}_n)]^T \mathbf{Q}_{n,k}^{-1}[\mathbf{z}_{n,k} - \tilde{\mathbf{h}}_{n,k}(\mathbf{x}_n)]\right), \tag{2.35}$$

which is (2.29) with $\mathbf{h}_{n,k}$ replaced by $\tilde{\mathbf{h}}_{n,k}$.

Note that in the additive Gaussian noise case the algorithm for particle filtering operates almost like in the general case. The only difference is that coefficients $\gamma_{n,k,r}$ can be calculated indirectly, without using a separate LS fitting, hence reducing the computational complexity. However in general coefficient obtained directly and indirectly will be different.

Polynomial Approximation

Using (2.22), it is obtained for (2.32)

$$\begin{aligned}\tilde{\mathbf{a}}_{n,k}(\mathbf{x}_n) = \tilde{\mathbf{h}}_{n,k}(\mathbf{x}_n) &= \sum_{\mathbf{r}=\mathbf{0}}^{R_p} \alpha_{n,k,\mathbf{r}} \prod_{m=1}^{M} x_{n,m}^{r_m} \\ &= \sum_{r_1=0}^{R_p} \cdots \sum_{r_M=0}^{R_p} \alpha_{n,k,r_1,\ldots,r_M}\, x_{n,1}^{r_1} \cdots x_{n,M}^{r_M}.\end{aligned} \tag{2.36}$$

Inserting this into (2.33) yields

$$\begin{aligned}\tilde{d}_{n,k}(\mathbf{x}_n) &= \sum_{\mathbf{r}=\mathbf{0}}^{2R_p} \gamma_{n,k,\mathbf{r}} \prod_{m=1}^{M} x_{n,m}^{r_m} \\ &= \sum_{r_1=0}^{2R_p} \cdots \sum_{r_M=0}^{2R_p} \gamma_{n,k,r_1,\ldots,r_M}\, x_{n,1}^{r_1} \cdots x_{n,M}^{r_M},\end{aligned} \tag{2.37}$$

with

$$\gamma_{n,k,r_1,\ldots,r_M} = \frac{1}{2}\sum_{r_1'=0}^{R_p} \cdots \sum_{r_M'=0}^{R_p} \sum_{r_1''=0}^{R_p} \cdots \sum_{r_M''=0}^{R_p} \alpha_{n,k,r_1',\ldots,r_M'}^T \mathbf{Q}_{n,k}^{-1} \alpha_{n,k,r_1'',\ldots,r_M''}, \tag{2.38}$$

with the condition on the indexes

$$(r_1', \ldots, r_M') + (r_1'', \ldots, r_M'') = (r_1' + r_1'', \ldots, r_M' + r_M'') = (r_1, \ldots, r_M). \tag{2.39}$$

Next, inserting expressions (2.36) and (2.37) into (2.17), we obtain

$$\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n) = \sum_{k=1}^{K} \sum_{\mathbf{r}=\mathbf{0}}^{2R_p} \beta_{n,k,\mathbf{r}}(\mathbf{z}_{n,k}) \prod_{m=1}^{M} x_{n,m}^{r_m}, \tag{2.40}$$

with

$$\beta_{n,k,\mathbf{r}} = \begin{cases} \alpha_{n,k,\mathbf{r}}^T \mathbf{b}_{n,k}(\mathbf{z}_{n,k}) - \gamma_{n,k,\mathbf{r}}, & \mathbf{r} \in \mathcal{R}_1 \\ -\gamma_{n,k,\mathbf{r}}, & \mathbf{r} \in \mathcal{R}_2, \end{cases} \tag{2.41}$$

where $\mathcal{R}_1$ is the set of all $\mathbf{r} = (r_1 \cdots r_M) \in \{0, \ldots, R_p\}^M$ such that $\sum_{m=1}^{M} r_m \leq R_p$ and $\mathcal{R}_2$ is the set of all $\mathbf{r} \in \{0, \ldots, 2R_p\}^M \setminus \mathcal{R}_1$ such that $\sum_{m=1}^{M} r_m \leq 2R_p$. Finally,

changing the order of summation in (2.40) gives

$$\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n) = \sum_{\mathbf{r}=\mathbf{0}}^{2R_p} B_{n,\mathbf{r}}(\mathbf{z}_n) \prod_{m=1}^{M} x_{n,m}^{r_m} \tag{2.42}$$

with

$$B_{n,\mathbf{r}}(\mathbf{z}_n) = \sum_{k=1}^{K} \beta_{n,k,\mathbf{r}}(\mathbf{z}_{n,k}) \tag{2.43}$$

The coefficients $B_{n,\mathbf{r}}(\mathbf{z}_n)$ can again be calculated using a consensus algorithm. For each time n, the number of consensus algorithm that have to be executed in parallel, is given by $N_c = \binom{2R_p+M}{2R_p} - 1$ (the -1 because the coefficient for $\mathbf{r} = \mathbf{0}$ corresponds to the JLF factor that does not depend on $\mathbf{x}_n$)

2.4.4 LC-DPF Algorithm

The distributed implementation of the particle filter algorithm is based on the SIR Filter discussed in (2.3.2), with the main difference that each sensors acts as a fusion center of a centralized PF. More specifically, sensor k tracks a particle representation of the global posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ using a *local PF* which can be used to compute the state estimate $\hat{\mathbf{x}}_{n,k}$ at time n based on the past and current measurement of *all* sensors. This requires each sensor to know the JLF $f(\mathbf{z}_n)|\mathbf{x}_n)$ as a function of $\mathbf{x}_n$ for the weight evaluation. Therefore, an approximation of the JLF is provided at each sensor via a distributed consensus algorithm. The algorithm can be summarized as follows.

LC-Based Distributed Particle Filter [8] At time $n = 0$, J particles $\mathbf{x}_{0,k}^{(j)}$, $j = 1, \ldots, J$ are sampled from a prior distribution $f(\mathbf{x}_0)$ at sensor k, for all k simultaneously. At any give time $n \geq 1$, the following steps are performed:

1. For each previous particle $\mathbf{x}_{n-1,k}^{(j)}$, a new particle $\bar{\mathbf{x}}_{n,k}^{(j)}$ is drawn from $f(\mathbf{x}_n|\mathbf{x}_{n-1,k}^{(j)})$.

2. The approximate JLF $\tilde{f}(\mathbf{z}_n|\mathbf{x}_n)$ is computed in a distributed way by means of the SLC as described in (2.4.2). Communication among neighboring sensors is required only on this step.

3. The weight associated with particles $\bar{\mathbf{x}}_{n,k}^{(j)}$ are calculated as

$$w_{n,k}^{(j)} = \frac{\tilde{f}(\mathbf{z}_n|\bar{\mathbf{x}}_{n,k}^{(j)})}{\sum_{j'=1}^{J} \tilde{f}(\mathbf{z}_n|\bar{\mathbf{x}}_{n,k}^{(j')})}, \quad j = 1, \ldots, J. \tag{2.44}$$

   This is evaluated for all particles $\bar{\mathbf{x}}_{n,k}^{(j)}$.

4. From the weighted particles $\{\bar{\mathbf{x}}_{n,k}^{(j)}, w_{n,k}^{(j)}\}_{j=1}^{J}$, an approximation of the MMSE state estimate can be computed as

$$\hat{\mathbf{x}}_{n,k} = \sum_{j=1}^{J} w_{n,k}^{(j)} \bar{\mathbf{x}}_{n,k}^{(j)}. \tag{2.45}$$

5. The set of particles and weights $\{\bar{\mathbf{x}}_{n,k}^{(j)}, w_{n,k}^{(j)}\}_{j=1}^{J}$ representing the current global posterior $f(\mathbf{x}_n|\mathbf{z}_{1:n})$ is *resampled*; this produces J resampled particles $\mathbf{x}_{n,k}^{(j)}$ with

identical weights. The $\mathbf{x}_{n,k}^{(j)}$ are obtained by sampling with replacement from the set $\{\bar{\mathbf{x}}_{n,k}^{(j')}\}_{j'=1}^{J}$, where $\bar{\mathbf{x}}_{n,k}^{(j')}$ is sampled with probability $w_{n,k}^{(j')}$

Now that we had an overview of the mathematical process behind distributed particle filtering, we can proceed to contextualize the formulæin our case study in the next chapter, while in chapter 4 an overview to the software implementation is given.

Chapter 3

Target Tracking via Particle Filtering

In this chapter we firstly define a specific scenario and the we try to adapt the knowledge acquired in chapter 2 to it, in order to have a better understanding of what we need to do later on in the software implementation.

3.1 Case Study: Target Tracking Scenario

Target tracking is one application of the more generic method of *state tracking*, where the state of the world that needs to be tracked is the state describing a target object (or many objects), and the state vector usually describes the object position and speed expressed in a certain measurement system, that has to be the same as the one used to describe the world in which the object moves. With this brief description we can already identify some important elements that defines our case study.

First of all we assume that an object is moving in a limited 2-dimensional space and its real state vector describes the two position coordinates, and the relative speeds along each axis which at a discrete time n assumes the form:

$$\mathbf{x}_n = \{x_n, y_n, \dot{x}_n, \dot{y}_n\} \tag{3.1}$$

and the *system model* that defines the state evolution simply applies linear motion laws over time affected by driving zero-mean Gaussian noise:

$$\mathbf{g}_n(\mathbf{x}_{n-1}, \mathbf{u}_n) = \begin{cases} x_n &= x_{n-1} + \dot{x}_{n-1} + u_{x,n} \\ y_n &= y_{n-1} + \dot{y}_{n-1} + u_{y,n} \\ \dot{x}_n &= \dot{x}_{n-1} + u_{\dot{x},n} \\ \dot{y}_n &= \dot{x}_{n-1} + u_{\dot{y},n} \end{cases} \tag{3.2}$$

Now, in order to apply particle filtering, a sensing element is required and for that we assume the environment to be populated with located sensors capable of detecting the object and periodically sample a measurement that depends on the real object state vector $\mathbf{x}_n$. The measurement assumed is scalar and it's the *Euclidean distance* between the position of the sensor and the one of the object, and the measurement function for the sensor k has the following form:

$$h_{n,k}(\mathbf{x}_n) = h_{n,k}(x_n, y_n) = \sqrt{(x_n - x_k)^2 + (y_n - y_k)^2} \tag{3.3}$$

where x_k and y_k are the coordinates of the position of the sensor, which we assume locally known at the sensor. The measurement process yields a value that is affected by zero-mean additive Gaussian noise which makes us fall into the Gaussian noise

special case (cfr. 2.4.3), and the scalar value obtained by sensor k at discrete time n is:

$$z_{n,k} = h_k(x_n, y_n) + v_{n,k} \tag{3.4}$$

where $v_{n,k} \sim \mathcal{N}(0, \sigma_v^2)$. Note that the measurement $z_{n,k}$ doesn't depend on the velocities $\dot{x}_n$ and $\dot{y}_n$. Finally the local likelihood function and the JLF assume the form:

$$f(z_{n,k}|\mathbf{x}_n) = \frac{1}{\sqrt{2\pi\sigma_v^2}} \exp\left(-\frac{1}{2\sigma_v^2}[z_{n,k} - h_{n,k}(\mathbf{x}_n)]^2\right) \tag{3.5}$$

$$f(\mathbf{z}_n|\mathbf{x}_n) = \frac{1}{\sqrt{(2\pi\sigma_v^2)^K}} \exp\left(-\frac{1}{2\sigma_v^2}\sum_{k=1}^{K}[z_{n,k} - h_{n,k}(\mathbf{x}_n)]^2\right) \tag{3.6}$$

and hence

$$S_n(\mathbf{z}_n, \mathbf{x}_n) = \frac{1}{\sigma_v^2}\sum_{k=1}^{K} h_{n,k}(\mathbf{x}_n)\left[z_{n,k} - \frac{1}{2}h_{n,k}(\mathbf{x}_n)\right] \tag{3.7}$$

3.2 Particle Filtering in the Case Study

After defining the elements to work with, we now take a look at how the mathematical particle filtering formulæ are adapted to the case study.

3.2.1 LC Approximation

For LC, the measurement function $h_{n,k}(\mathbf{x}_n)$ is approximated by a polynomial of degree $R_p = 2$ as follows (cfr. 2.36)

$$h_{n,k}(\mathbf{x}_n) = h_{n,k}(x_n, y_n) \approx \tilde{h}_{n,k}(x_n, y_n) = \sum_{r_1=0}^{2}\sum_{r_2=0}^{2} \alpha_{n,k,r_1,r_2} x_n^{r_1} y_n^{r_2} \tag{3.8}$$

note that $M = 2$ because the velocities are not tracked, since the sensors can only sample a measurement dependent on the position coordinate. This approximation requires a total of $\binom{R_p+M}{R_p} = \binom{2+2}{2} = 6$ coefficients to be computed via regression and the expanded polynomial looks like the following:

$$\tilde{h}_{n,k}(x_n, y_n) = \alpha_{n,k,0,0} + \alpha_{n,k,1,0}x_n + \alpha_{n,k,0,1}y_n + \alpha_{n,k,1,1}x_n y_n + \alpha_{n,k,2,0}x_n^2 + \alpha_{n,k,0,2}y_n^2 \tag{3.9}$$

Using $\tilde{h}_{n,k}(\mathbf{x}_n)$, $\tilde{d}_{n,k}(\mathbf{x}_n)$ is approximated as

$$\tilde{d}_{n,k}(\mathbf{x}_n) = \sum_{r_1=0}^{4}\sum_{r_2=0}^{4} \gamma_{n,k,r_1,r_2} x_n^{r_1} y_n^{r_2} \tag{3.10}$$

with

$$\gamma_{n,k,r_1,r_2} = \frac{1}{2}\sum_{r_1'=0}^{2}\sum_{r_2'=0}^{2}\sum_{r_1''=0}^{2}\sum_{r_2''=0}^{2} \alpha_{n,k,r_1',r_2'}(\sigma_v^2)^{-1}\alpha_{n,k,r_1'',r_2''} \tag{3.11}$$

with the constraint $r_1' + r_1'' = r_1$ and $r_2' + r_2'' = r_2$. This for a fixed pair (r_1, r_2) leads to select all the elements of the half-matrix of coefficients $A_{n,k}$ along a single

diagonal:

$$A_{n,k} = \begin{pmatrix} \alpha_{n,k,0,0} & \alpha_{n,k,0,1} & \alpha_{n,k,2,0} \\ \alpha_{n,k,1,0} & \alpha_{n,k,1,1} & 0 \\ \alpha_{n,k,0,2} & 0 & 0 \end{pmatrix}. \tag{3.12}$$

Finally putting everything together we obtain

$$\tilde{S}(\mathbf{z}_n, \mathbf{x}_n) = \sum_{k=1}^{K} \sum_{r_1=0}^{4} \sum_{r_2=0}^{4} \beta_{n,k,r_1,r_2}(z_{n,k}) x_n^{r_1} y_n^{r_2} \tag{3.13}$$

where β_{n,k,r_1,r_2} assumes the following values (cfr. 2.41)

$$\beta_{n,k,r_1,r_2}(z_{n,k}) = \begin{cases} \alpha_{n,k,r_1,r_2}(\sigma_v^2)^{-1} z_{n,k} - \gamma_{n,k,r_1,r_2}\,, & (r_1, r_2) \in \mathcal{R}_1 \\ -\gamma_{n,k,r_1,r_2}\,, & (r_1, r_2) \in \mathcal{R}_2, \end{cases} \tag{3.14}$$

where $\mathcal{R}_1$ is the set of pairs $(r_1, r_2) \in \{0,1,2\}^2$ such that $r_1 + r_2 \leq 2$ (cfr. matrix $A_{n,k}$ in (3.12)) and $\mathcal{R}_2$ is the set of pairs $(r_1, r_2) \in \{0,1,2,3,4\}^2 \setminus \mathcal{R}_1$ such that $r_1 + r_2 \leq 4$. Note that in case of $\mathcal{R}_2$ elements, the reader has to refer to equation (3.11) which, for elements with index $r_i > 2$, due to the constraint, only requires elements present in $A_{n,k}$. Now the set of β_{n,k,r_1,r_2} is ready to be computed locally and then approximated via a consensus algorithm. For easiness of reading the resulting $B_{n,k}$ matrix containing $\binom{2R_p+M}{2R_p} = \binom{4+2}{4} = 15$ coefficients is

$$B_{n,k} = \begin{pmatrix} \beta_{n,k,0,0} & \beta_{n,k,0,1} & \beta_{n,k,0,2} & \beta_{n,k,0,3} & \beta_{n,k,0,4} \\ \beta_{n,k,1,0} & \beta_{n,k,1,1} & \beta_{n,k,1,2} & \beta_{n,k,1,3} & 0 \\ \beta_{n,k,2,0} & \beta_{n,k,2,1} & \beta_{n,k,2,2} & 0 & 0 \\ \beta_{n,k,3,0} & \beta_{n,k,3,1} & 0 & 0 & 0 \\ \beta_{n,k,4,0} & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{3.15}$$

Along with the approximation needed for $\tilde{S}(\mathbf{z}_n, \mathbf{x}_n)$, the approximation of the normalization factor $c_{n,k}(z_{n,k})$ (cfr. 2.30) is needed and for our case study it assumes the form :

$$c_{n,k}(z_{n,k}) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2} z_{n,k}^2\right) \tag{3.16}$$

which computed over all sensors becomes (2.13) and rewritten as a logarithm, it is ready to be estimated globally via consensus

$$\log C_n(z_n) = \sum_{k=1}^{K} \log c_{n,k}(z_{n,k}) = \sum_{k=1}^{K} \log\left[\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2} z_{n,k}^2\right)\right]. \tag{3.17}$$

Note that for any time n the measurement vector $\mathbf{z}_n$ is a constant, so $c_{n,k}(z_{n,k})$ is firstly computed at each sensor and then made into a logarithm for the consensus algorithm.

This concludes all the necessary approximations needed before estimating the global JLF via consensus.

3.2.2 Consensus steps

Now that we have a clearer idea on how the set of coefficients than needs to be estimated via consensus look like, we can now take a look at the consensus algorithm steps, adapted to the case study.

At s sensor k the following steps are performed:

1. The set of coefficient $\{\beta_{n,k,r_1,r_2}\}_{(r_1,r_2)\in\{0,\dots,4\}^2}$ and the normalization factor $c_{n,k}(z_{n,k})$ are computed as shown in section 3.2.1;
2. *Dynamic consensus algorithm* (single step) - for each non-zero element in $B_{n,k}$:
   (a) if $n = 1$, initialize the internal state as $s_{0,k,r_1,r_2} = \beta_{1,k,r_1,r_2}$
   (b) compute the internal state (cfr. 2.24)
   $$\zeta_{n,k,r_1,r_2} = \mu s_{n-1,l,r_1,r_2} + (1-\mu)\beta_{n,k,r_1,r_2} \tag{3.18}$$
   (c) ζ_{n,k,r_1,r_2} is broadcast to all neighboring sensors $k' \in \mathcal{N}_k$
   (d) The new internal state s_{n,k,r_1,r_2} is calculated in a single consensus step as in (2.25) and the weights in (2.26)
   (e) $\tilde{\beta}_{n,r_1,r_2}$ is obtained by scaling s_{n,k,r_1,r_2} by K, which is the total number of sensors
   $$\tilde{\beta}_{n,r_1,r_2} = K s_{n,k,r_1,r_2} \tag{3.19}$$
3. The same process is performed for the logarithm of the normalization factor $\log c_{n,k}(z_{n,k})$ so to obtain the internal state $\varsigma_{n,k}$ which once scaled by the number of sensors K, becomes the logarithm of the global normalization factor $\log \tilde{C}_n(z_n)$;

After turning the estimate of the logarithm of the normalization factor back into $\tilde{C}_n(z_n)$, the final result of the JLF estimation has the following form

$$\tilde{f}(z_n|\mathbf{x}_n) = \tilde{C}_n(z_n)\exp(\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n)) \tag{3.20}$$

where

$$\tilde{S}_n(\mathbf{z}_n, \mathbf{x}_n) = \sum_{r_1=0}^{4}\sum_{r_2=0}^{4}\tilde{\beta}_{n,r_1,r_2}(\mathbf{z}_n)x_n^{r_1}y_n^{r_2}. \tag{3.21}$$

Note that $\tilde{\beta}_{n,r_1,r_2}$ has already been scaled by the number of sensors K at the end of the consensus algorithm and it does now not depend on k.

3.2.3 The PF steps

Using the adapted elements defined in this chapter we now take a look at the particle filtering algorithm to give a better vision on how the formulæ are applied.

At time $n = 0$, a set of J particles $\mathbf{x}_{0,k}^{(j)} = (x, y, \dot{x}, \dot{y})_{0,k}^{(j)}$, $j = 1, \dots, J$ is sampled from a prior distribution. For the position hypothesis we chose 2 different distribution function combinations based on 2 cases:

(i) The sensor samples a measurement of the object before initialization,

(ii) The sensor samples the first measurement only after initialization.

On case (i) a *Uniform distribution function* $\mathcal{U}(l, u)$ for each coordinate of the position is used, l and u are the lower and upper bounds for the functions respectively, which in our case coincide with the coordinates of the left-to-right bounds coordinates for the x and the top-to-bottom bounds coordinates for the y. This way the sensor creates a cloud of particles uniformly spread all over the virtual space, successive iterations should filter the particles keeping only the ones close to real object position.

On case (ii) the sensor performs a measurement before generating the particles so it already has an idea at which distance the object may be located. With this information available it is possible to generate a cloud of particles around the sensor, around the distance sampled, according to a zero-mean *Gaussian distribution* $\mathcal{N}(0, \sigma^2)$ which adds some noise to the sampled measurement in order to create variety among the guessed particles. To decide at which angle around the sensor position the particle should be located, again an *Uniform distribution* is used, with bounds $[0, 360]$. Once the guessed distance and the angle are set, they are converted into Cartesian coordinates with a simple trigonometric formula

$$\begin{cases} x_{0,k} &= r\ cos\,\phi \\ y_{0,k} &= r\ sin\,\phi \end{cases} \tag{3.22}$$

where $r = z_{0,k} + n_{0,k}$, with $n_{0,k} \sim \mathcal{N}(0, \sigma^2)$ is the varied distance and ϕ is the sampled angle.

Once the initial J particles are generated, the particle filter can start performing the iterative tracking process.

1. *Prediction*: Here the function of the system model (3.2) is applied to all particles $\mathbf{x}_{n-1,k}^{(j)}$, generating J predicted particles $\bar{\mathbf{x}}_{n,k}^{(j)} = (\bar{x}_{n,k}^{(j)}, \bar{y}_{n,k}^{(j)})$;

2. *JLF Computation*: the J predicted particles and the measurement function $h_{n,k}(\cdot)$ are used in the regression to approximate $\tilde{h}_{n,k}(\cdot)$ and consequently estimate the JLF via consensus algorithm as described in section 3.2.2;

3. *Update*: Using the approximate JLF, the non-normalized weight for each predicted particle is computed as follows

$$\bar{w}_{n,k}^{(j)} = \tilde{C}_n(z_n)\exp(\tilde{S}_n(z_n, x_n, y_n)) \tag{3.23}$$

   where $\tilde{S}_n(z_n, x_n, y_n)$ fully expanded has the form

$$\begin{aligned} &\sum_{r_1=0}^{4}\sum_{r_2=0}^{4}\tilde{\beta}_{n,r_1,r_2}(\mathbf{z}_n)x_n^{r_1}y_n^{r_2} = \\ &\beta_{n,k,0,0} + \beta_{n,k,0,1}y + \beta_{n,k,0,2}y^2 + \beta_{n,k,0,3}y^3 + \beta_{n,k,0,4}y^4 + \\ &\beta_{n,k,1,0}x + \beta_{n,k,1,1}xy + \beta_{n,k,1,2}xy^2 + \beta_{n,k,1,3}xy^3 + \beta_{n,k,2,0}x^2 + \\ &\beta_{n,k,2,1}x^2y + \beta_{n,k,2,2}x^2y^2 + \beta_{n,k,3,0}x^3 + \beta_{n,k,3,1}x^3y + \beta_{n,k,4,0}x^4 \end{aligned} \tag{3.24}$$

   The weight $\bar{w}_{n,k}^{(j)}$ is the normalized by the sum of all J weights so to obtain $w_{n,k}^{(j)}$.

4. *MMSE Estimate*: The global MMSE estimate is computed for each predicted element of the state vector involved in the tracking over all J particles:

$$\hat{x}_{n,k} = \sum_{j=1}^{J} w_{n,k}^{(j)}\bar{x}_{n,k}^{(j)} \tag{3.25}$$

$$\hat{y}_{n,k} = \sum_{j=1}^{J} w_{n,k}^{(j)}\bar{y}_{n,k}^{(j)} \tag{3.26}$$

5. *Resampling*: The J particle pairs $(\bar{x}_{n,k}^{(j)}, \bar{y}_{n,k}^{(j)})$ are sampled with replacement so to create a new set of J pairs $(x_{n,k}^{(j)}, y_{n,k}^{(j)})$. The probability for each pair j to be chosen is $w_{n,k}^{(j)}$. In the chosen resampling method *Monte Carlo Importance Sampling* a *discrete cumulative distribution function* is computed using all J weights and for each new element that needs to be extracted an *Uniform distribution* $u_{n,k} \sim \mathcal{U}(0,1)$ is sampled and the *bin* in which the sample is included, defines the index j of the particle that has to be added to the resampled set.

This concludes contextualization of the mathematical knowledge acquire in chapter 2 and should now allow us to continue to chapter 4 where a more architectural definition of the problem is given and the implementation of steps discussed above into the simulator software is described.

Chapter 4

Software Application

In this chapter we take Data Fusion theory and the particle filtering knowledge discussed in chapter 2 and put them into practice on the case study scenario discussed in chapter 3 focusing on the software implementation of a Target Tracking Simulator application structured as a Java/AgentSpeak multi agent system.

4.1 Software Architecture for the Case Study

In chapter 3 we defined some central elements that can be directly translated into the architecture of our software implementation. The generic idea of this project is to create a piece of software that is able to simulate an object moving around in a virtual 2-dimensional space while being observed by a distributed network of located sensors. These sensors have limited perception capabilities and can only measure the distance between them and the moving object. The network of sensors has to employ distributed communication and particle filtering in order to correctly update the state that keeps track of the object position. The software should also be capable of showing some information on screen to the user, such as the position of the tracked object and of the sensors displaced in the 2-dimensional space. It then should also draw the result of the tracking in order to do at least qualitative evaluation.

4.1.1 The Approach

After a long time spent studying the literature around this novel practices for distributed systems, we decided to employ Java and the AgentSpeak library Jason in our project for the following reasons:

- Java is a well known Object Oriented Language that allows a lot of flexibility.
- The Jason library is available for Java which allows the integration with AgentSpeak.
- The agent oriented, event-driven infrastructure supplied by Jason is easy enough to allow the programmer to focus on the core matters of the project, instead of dealing with agent-related issues in the code. This allows an easy deployment of a distributed agent system.
- Jason supports Internal Actions that can be written fully in Java language, allowing the creation of custom logic that can easily be executed by the BDI agent.
- AgentSpeak allows a quick and easy way to define the behavior of a BDI agent.

For the development of the project, we decided to take a staged approach which is described as high level requirements as follows.

1. First of all we had to set a working base for the moving object that has to be tracked. The object needs to be controlled by the user and that means controlling the position where it enters the environment and its movement during the simulation. The UI and the system model are also defined in this stage. This sets the structure where in the following stages the sensor agents are inserted.

2. Once the Environment is able to host a fully controllable moving object, it's time to define the logic for the sensors performing the tracking. We decided to approach their introduction in two sub-stages. The first, and more simple one, is done mainly to test the centralized version of the particle filtering algorithm, while the second is done to implement a fully distributed system of local particle filters.

   (a) On this stage, in order to try the centralized algorithm of the Particle filter, a FA-based particle filter is created, where simplified sensor agents only perform measurements on the moving object from their relative position in the environment and send the data to a Fusion Agent where the centralized particle filter is implemented. This "middle of the road" experiment allows us to understand better the PF algorithm once it has been translated into Java code and sets a base structure for the fully distributed stage.

   (b) Now the LC-based distributed particle filter is implemented. Each sensor now performs a local version of the centralized particle filter with the difference that this time the likelihood function is computed with a distributed consensus algorithm. This way the resilience of the system against failures is improved while maintaining a relatively low complexity and low volume of transmitted data, since instead of sending all the measurements collected in the network to a single fusion center, each sensor needs only to communicate with its own neighbors. The structure of the code explored in the first stage is expanded and adapted to accommodate the various changes.

3. Once the core logic of the project is working, it is possible to optimize the code and add various quality-of-life features.

Note that the Environment is also adjusted at each stage while maintaining all the features introduced in a previous stage.

4.1.2 Architecture for each stage

Stage 1

The first stage has no active sensor, and the only elements present are:

- The **Environment** element hosts the various agents in a virtual 2-dimensional space limited by borders and deals with all the interactions they would have with a real environment while also providing a simple way to show the information to the user through a GUI. In order to do this it needs to collect and model critical information such as the position of each agent in the virtual space. If we assume that anything outside an agent is part of the environment,

user commands also are, so the environment should facilitate agents' control by the user.

- The **moving object**. This is a very simple agent that can enter the environment and move around at any direction in the 2-dimensional space, or stay still. Its movement is controlled by the user and it can move forward, backwards or rotate (anti)clockwise at constant speed. The user is also able to decide the initial position of the object.

Stage 2: Introducing the sensors

In this stage we introduce the sensor agents and implement particle filtering in two sub-stages:

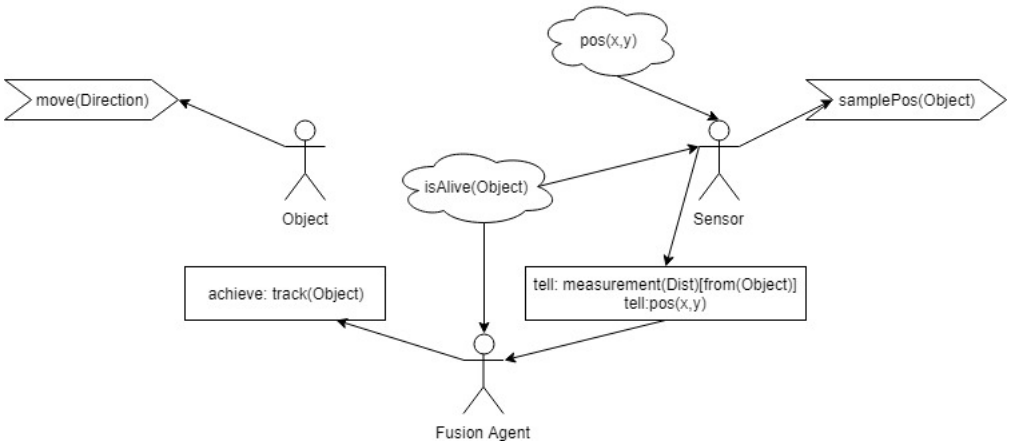

FIGURE 4.1: Agent configuration for stage (2.a). Note that behavior implemented for secondary operations is not shown in the diagram.

Fusion Agent method This sub-stage is solely intended to test the PF tracking algorithm in its centralized form, getting accustomed with the process and the coding tools needed, and set the base for the step towards a complete distributed version. The new elements are introduced as follows and are pictured in the diagram in figure 4.1.

- The **Sensor Agent** has a very simple behavior. Once positioned in the environment it cannot move and can only be removed by the user. We assume the sensor is able to know its position in the environment. When the trackable object enters the virtual space, the sensor percepts its presence and begins sampling the distance between itself and the object. Once the information is acquired, a message to the fusion agent is sent, containing the fresh measurement and the position of the sensor.

- The **Fusion Agent** doesn't need to be positioned in the environment as its only role is to collect the measurements of all sensors and to perform the particle filtering algorithm. In order to start the process, it needs to know when the trackable moving object enters the environment so to begin the tracking. If no sensor is active, the FA should not be able to initiate the PF algorithm.

The environment is modified so it can provide the right percepts to the agents, the correct response to the action *sample*, while tracking and showing to the user all

the new generated information (sensor positions, estimated object position...). The user should have access to a set of commands to manually or randomly position the sensor agents around the virtual space.

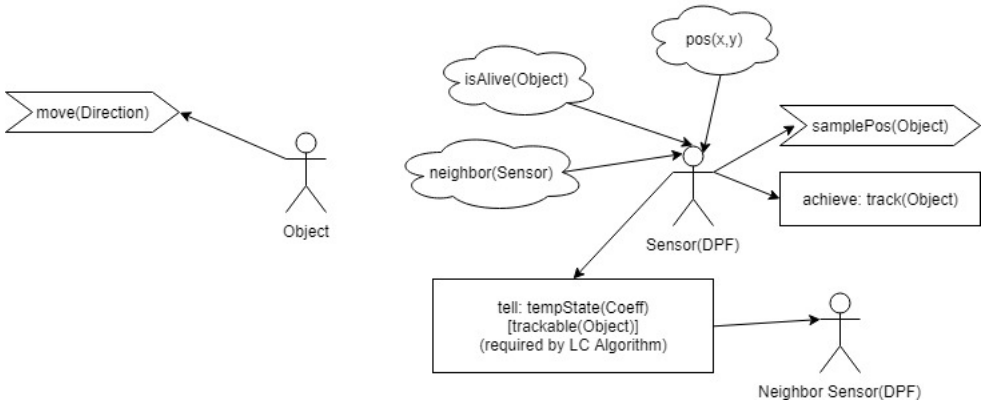

FIGURE 4.2: Agent configuration for stage (2.a). Note that behavior implemented for secondary operations is not shown in the diagram.

Distributed Consensus method With this configuration all the PF computation is transferred into the sensor agent, and the only communication happening is among sensors and it is during the consensus algorithm. Beside that, the sensor needs to know some information that we assume being provided by algorithms that are not the focus of this work. In this case the information needed is provided as percept by the environment, which is the major change to the environment element from the previous iteration.

4.2 Software Development

For the software implementation it was decided to use a combination of Java and AgentSpeak, respectively for the more procedural part and for the distributed agent operations and communications.

The Jason library [1] described in chapter 2, offers a good interface between Java and AgentSpeak especially because it offers a fair amount of access to agent's state and controls from the Java code, while also allowing the definition of *Internal Actions* written completely in Java, which, for more procedural operations, offer more flexibility compared to AgentSpeak.

For the more complex mathematical operations, e.g. regression or anything not included in the standard Math class in Java, the Apache Commons Mathematics Library [2] is used.

4.2.1 Implementing the Environment

Our custom Environment class implements the Jason Environment interface and it acts as the entry point for the agent system. The class should implement the method *executeAction* in order to apply changes to the environment induced by the agents.

```
[...]executeAction(String agName, Structure action){
 switch (action.getFunctor()) {
  case "action_name":{
    //logic for action
    //returns true if action is successful
    //otherwise returns false
  }
 }
}
```

An action doesn't necessarily need to trigger a perception change in the agent's belief base. In the set of actions that we implemented, the only one that updates the sensor's percepts is *sample*, which is meant to measure the distance between the sensor and the moving object at a certain time. When the action is called in the agent's AgentSpeak code, the underlying MAS infrastructure calls the method *executeAction* in the custom Environment class, passing some information such as the caller's name, the action's name and its eventual parameters. A simple case switch is implemented in order to route the correct logic depending on the action that needs to be executed. In the specific case of *sample*, the id of the agent is extracted from the action and the switch leads the control flow to enter the logic for the action. At this point the id of the object is extracted from the action parameters and both the positions of the sensor and of the object are retrieved from the world model and the distance between the two is computed. The result is then formatted into a literal and the method to update the caller agent's percept is performed.

```
//inside executeAction switch...

case "sample":

//retrieve the kind of measurement (distance)

String type = action.getTerm(0).toString();
```

```
//retrieve the target

String trackable = action.getTerm(1).toString();

//retrieve states from the model

TrackableState target = model.getState(trackable);
TrackableState sensor = model.getState(agName);

//measure distance affected by additive Gaussian noise

if (type.equals("distance")) {
 double distance =
  Measure.distanceXY(target.x, sensor.x, target.y, sensor.y)
   + noise.sample();
 distance = Math.max(distance, 0);

 //update sensor percepts

 removePerceptsByUnif(agName,
  literal("distance(_)[trackable(%s)]",
   trackable));
 addPercept(agName,
  literal("distance(%s)[trackable(%s)]",
   distance, trackable));
 informAgsEnvironmentChanged(agName);
 return true;
}
return false;
```

Note that agent perception can be modified and updated in other occasions, for example when a new sensor is positioned in the environment, its neighbors need to be informed, or when the object enters the environment, all sensors need to perceive it. All of this happens in the method to create new agent which is called with the press of a button in the UI and an*ActionListener* on that UI element. The *actionPerformed* method routes the action with a switch and the creation of a new sensor agent is described as follows.

```
[...]
case "spawn_sensorDPF":

 //the new sensor agent is created and is given a name
 //and it is specified what logic it needs to has (.asl file)

 String newAgentName =
  getEnvironmentInfraTier().getRuntimeServices().createAgent(
   "sensorDPF" + sensorNumber++,
    "sensorDPF.asl",
     null, null, null, null, null);

 //the new agent is started
```

```
 getEnvironmentInfraTier()
  .getRuntimeServices()
   .startAgent(newAgentName);

 //at the same time it is added to the model
 //note that even if an update method is used
 //a new entry is added to the model
 //if an agent with the same name doesn't exist
 //the initial sensor position is retrived from the UI

 model.updateState(newAgentName,
  view.getSettings().getSensorInitState());

 //update the total_sensors belief for all sensors

 totalDPFSensors++;
 for (String agent : getEnvironmentInfraTier()
  .getRuntimeServices()
   .getAgentsNames()) {
  if (agent.startsWith("sensorDPF")) {
   removePerceptsByUnif(agent,
    literal("total_sensors(_)"));
   addPercept(agent,
    literal("total_sensors(%s)", totalDPFSensors));
   informAgsEnvironmentChanged(agent);
  }
 }
break;
```

The custom environment class, beside dealing with agent interactions, needs also to maintain a system model for the virtual space, as to "remember" where each element is located, and all the information that could be perceived by the agents, including the user. Then, to show this information to the user, the class also needs to implement a GUI. For these secondary, but important task, we decided to implement two designated elements called *world model* and *view*.

World Model The world model is a class designated to store and update the information relative to the "visible" state of the agent in the environment, which in our case is the array of values that describes one agent's position, speed and orientation in the 2-dimensional space. The model is time-dependent and periodically updates the state of moving objects according to linear motion laws.

This class also defines an *observer interface* which is implemented by the custom environment and the view. This allows the two classes to execute certain operations whenever there is a change in the state of the model. For example whenever the object moves, it causes a change in the world model, in succession the latter signals the custom environment class and the view that the state update operation was successful. This triggers the environment class to update the percepts at the sensors, and the view to update the position of the object on screen.

```
//the interface for the observers

public interface WorldObserver {
```

```
void onStateChanged(String trackableID,
 TrackableState newTrackableState,
  TrackableState oldTrackableState);

void onStateRemoved(String trackableID,
 TrackableState removedState);
}

//methods informing the observers that the model has changed

private synchronized void notifyStateChanged(String
   trackableID,
 TrackableState newTrackableState,
  TrackableState oldTrackableState) {
observers.forEach(o -> o.onStateChanged(trackableID,
 newTrackableState, oldTrackableState));
}

private synchronized void notifyStateRemoved(
 String trackableID,
  TrackableState removedState) {
observers.forEach(o -> o.onStateRemoved(trackableID,
 removedState));
}

//methods for adding and removing states
//world borders are taken in consideration
//so agents can't be positioned outside

public synchronized void updateState(String id,
 TrackableState newState) {
 if (inBorders(newState)) {
  TrackableState old = trackableStates.put(id, newState);
  notifyStateChanged(id, newState, old);
 }
}

public synchronized void removeState(String id) {
 TrackableState removed = trackableStates.remove(id);
 notifyStateRemoved(id, removed);
        }
```

View This is the class designated to draw the GUI and all the necessary elements on screen for the user. Within the GUI a set of buttons and controls are available to the user which allow to position the object and the sensors around the virtual space. To make the qualitative evaluation of the tracking possible, the GUI also has to draw either the particle cloud or the MMSE computed by each particle filter (FA or sensors). In order to do this a container class is implemented which acts like a small database, storing the position of each particle grouped by sensor and then by object (in the case in the future tracking more than one object was necessary). The GUI uses

Java Swing to draw all the elements on a window which is periodically refreshed. At each refresh the object and sensors' position are consulted in the world model, the particle position are instead consulted in the custom container and everything is painted on screen.

These support elements of the custom Environment class are initialized in its constructor.

4.2.2 Implementing the Particle Filter Algorithm

We now explain how each step of the PF algorithm presented in section 2.4.4 was implemented in our application.

Initialization In the initialization step, happening right after a sensor perceives a trackable object entering the environment, and before the first iteration of the particle filtering algorithm, a fresh set of particles is initialized at the sensor and memorized inside its Belief Base via a dedicated Internal Action called by the agent itself once the event that perceives the object is triggered. The idea is to populate the set with hypothesis on the trackable object's position within a certain area according to a certain probability distribution. Two ways of initialization have been tested each using a different combination of pdfs:

- *Unknown initial location* In the first case we assume the trackable object could be anywhere inside a rectangle with fixed borders at the moment it enters the environment with uniform probability. For this reason a Uniform pdf is used and the class *UniformRealDistribution* from the Apache library is instantiated for each one of the position coordinates with the respective borders coordinates as min/max parameters.

- *Known initial location* In this case an initial measurement (with Gaussian noise) on the position of the object is taken before commencing the particles initialization. For this reason an Uniform distribution is used to sample the angle between 0 and 360 around the sensor at which the particle will be initialized. Then using the distance and the sampled angle a second 0-mean Gaussian pdf (class *NormalDistribution*) is used to add some variability in the distance for each different particle. This way a cloud of particles around the sensor is created possibly giving a bit more of accuracy in the early iterations of the PF algorithm, increasing the chance of a good tracking result. After each sample is taken, the position is translated into the common reference system of coordinates using the sensor position in the environment.

Each new particle is then formatted into a literal and stored into the agent's belief base using the appropriate Jason method call.

Particle update In the particle update step, each particle is updated according to the system model that defines the state change of the object with time. In this case the object is simply moving around with a certain speed and direction, but since the only part of the object state that is being tracked is the position, we limited the update process to change the object location adding Gaussian noise scaled to a balance factor using values present in the numerical example in [7] as a reference.

```
// update states according to state transition
```

```
double std_dev = 10;
NormalDistribution noise =
 new NormalDistribution(0, std_dev);
for (double[] s : states) {
 s[0] += s[2] + 0.5 * noise.sample();
 s[1] += s[3] + 0.5 * noise.sample();
 s[2] += Math.max(0,noise.sample());
 s[3] += Math.max(0,noise.sample());
}
```

where the state array is $s : (x, y, \dot{x}, \dot{y})$. Note that all values, such as the standard deviation, should be adjusted based on the dimension used in the system model in order to obtain a better tracking result.

Particle weighing This steps happens once the likelihood function is ready. The process where the likelihood function is computed is better explained in the dedicated section (4.2.2).

The result of the likelihood function computation is an object on which the method *weigh(s)* can be invoked. As input it receives an array containing the two position coordinates for the hypothetical object location stated by a particle. The output is the non normalized weight value associated for that particle. Each weight needs then to be normalized so that the sum of all weights is ≈ 1 (due to limited decimal representation of data types it may happen that sum is lower than 1 if some particles happen to be far from the likelihood position). To normalize each weight a sum of all weights is computed and used for normalization.

MMSE approximation For MMSE approximation an average value is computed for each coordinate of the state using the computed weights as weights in the mean expression 2.6, resulting in the centroid of the particle cloud. Note that each sensor computes its own MMSE approximation which may differ from the approximations of other sensors due to a different random set of particles.

Resampling Any resampling with replacement can be used in this step, so we decided to implement a simple *Monte Carlo Importance Sampling* algorithm in a method that accepts a list of the pairs $\{w_{n,k}^{(j)}, \mathbf{x}_{n,k}^{(j)}\}_{j=1}^{J}$ and the number of samples needed (J), and outputs a list of J sampled particles $\bar{\mathbf{x}}_{n,k}^{(j)}$. The algorithm computes a cumulative distribution function using the weights provided as input, by creating an array where each element is the sum of all the previous weights $j < i$ and the j-th weight, where i is the array index. This way a series of bins is created and a *UniformRealDistribution* object from the Apache library is instantiated between 0 and the last index of the array. J iterations are performed where a sample is taken from the Uniform Distribution and the bin where the sample "lands" is found. The index associated with that bin becomes the index of the particle that is sampled, which is then added to the list of sampled particles instantiated for the output.

```
[...]samplingMC(List<Pair<Double, E>> input, int maxSamples){
 List<E> samples = new LinkedList<>();
 double[] c = new double[input.size()];
 c[0] = input.get(0).getFirst();
 for (int i = 1; i < c.length; i++)
  c[i] = c[i - 1] + input.get(i).getFirst();
```

```
 UniformRealDistribution unif =
  new UniformRealDistribution(0, c[c.length - 1]);
 for (int j = 0; j < maxSamples; j++) {
  double u = unif.sample();
  int l = 0;
  for (; l < maxSamples && c[l] <= u; l++)
   ;
  samples.add(input.get(l).getSecond());
 }
 return samples;
}
```

Likelihood approximation and consensus

Depending on the configuration of the application a different way of computing the likelihood function is implemented.

FA configuration In the Fusion Agent configuration, each sensor that perceives the trackable object performs a measurement of the distance at regular intervals with a simple *while* iteration that is present in the goal triggered when the trackable object enters the environment. The measurement is then sent to the FA as a message, which stores the information in FA belief base.

```
[...]
+sensing(Object) <-
    while(sensing(Object)){
        .wait(10);
        sample(distance,Object)}.

//When the sensor obtains a new percept from the environment
//send the information to the fusion agent

+distanceFrom(Object,D) <-
    ?position(X,Y);
    .send(fa,tell,distanceFrom(Object,D)[pos(X,Y)]).
```

The FA is also informed when the trackable object enters the environment so to begin the particle filtering algorithm, which is the same that is performed at each sensor in the LC configuration, with the exception of the likelihood function computation.

```
[...]
//when the object enters the environment
+!track(Object) <-
        while(cycle(C)[trackable(Object)]){
         //Particle filtering IA
         pf.updateState(Object,C);
         //update counter for tracking iterations
         -+cycle(C+1)[trackable(Object)];
        }.
```

The FA, knowing all sensors' measurements and their respective positions, is able to compute the likelihood function in a centralized way as the intersection of all the

local likelihood functions of all sensors (cfr. 2.9), treating each function as a Gaussian pdf shifted by the measurement distance, which is also independent from all the other functions.

To perform the likelihood function computation, all the available sensors' measurements and relative positions are collected from the belief base and saved inside an instantiated object that we called *LikelihoodFunction*. This class features the method *weight(p)*, that at each call sequentially computes the product of all the local likelihood functions and outputs the weight value for the particle given as input. During a single iteration, a local likelihood function is computed as a Gaussian distribution centered at the value of the measurement (distance) $z_{n,k}$ and then sampled at the distance obtained between the sensor k and the input particle, using the same measurement model as the sensor.

LC configuration In the LC configuration, the behavior of the sensor agent is similar to the previous case, with the main difference that no measurement is sent to a FA. The major changes happens in the particle filtering Internal Action where this time the exponential basis function approximation is implemented. Before the actual consensus algorithm, the local likelihood approximation parameters need to be computed and, since our simulation fits the special case of Gaussian measurement noise (2.4.3), we applied that same process in Java code. Each sensor computes its own set of local parameters using the particles $\{\mathbf{x}_{n,k}^{(j)}\}_{j=1}^{J}$ and the measurement model function $h_k(\cdot)$. Firstly the set of $\alpha_{n,k,\mathbf{r}}$ coefficients is computed, in order to approximate the $h_k(\cdot)$. In order to do that, a regression class from the Apache library is used which takes in input the polynomial expansion computed at each particle position with the relative function $h_k(\cdot)$ result. The polynomial expansion is calculated with $R_p = 2$ and $M = 2$, so the expansion has the form of $x + y + xy + x^2 + y^2$ (cfr. 2.32) and each element of the regression input vector becomes each element of the polynom with that same order. Pairing the input vector with $h_k(x, y)$ as input of the regression object for each particle, yields the set of α_{n,k,r_1,r_2} parameters $\{\alpha_{n,k,0,0}, \alpha_{n,k,1,0}, \alpha_{n,k,0,1}, \alpha_{n,k,1,1}, \alpha_{n,k,2,0}, \alpha_{n,k,0,2}\}$, where the indexes r_1 and r_2 correspond to the exponent of the relative state element.

```
computeAlphas(List<double[]> states,
 Function<double[], Double> h) {
 UpdatingMultipleLinearRegression sr =
  new MillerUpdatingRegression(5, true);
 for (double[] state : states) {
  double[] v = { state[0], state[1] };
  sr.addObservation(polynomExpand2D(v),
   h.apply(state));
 }
 return sr.regress().getParameterEstimates();
}
```

where h is the function for the measurement model, given as parameter

```
Function<double[], Double> h = v -> {
 return Measure.distanceXY(v[0], posX, v[1], posY);
};
```

and *polynomExpand2D()* expands the 2D vector, into the needed 6-element expansion.

The result of this operation is a vector with $\binom{R_p+M}{R_p} = \binom{2+2}{2} = 6$ coefficients for the polynom approximating the measurement function $h_k(\cdot)$. In the next step the β coefficients need to be computed (cfr. 2.41). In this step the measurement $z_{n,k}$ is involved, so it gets acquired from the sensor's belief base. This time a new vector of $\binom{4+2}{4} = 15$ coefficients is computed and their order in the array is $\{\beta_{n,k,0,0}, \beta_{n,k,0,1}, \beta_{n,k,1,0}, \beta_{n,k,0,2}, \beta_{n,k,1,1}, \beta_{n,k,2,0}, \beta_{n,k,0,3}, \beta_{n,k,1,2}, \beta_{n,k,2,1}, \beta_{n,k,3,0}, \beta_{n,k,0,4}, \beta_{n,k,1,3}, \beta_{n,k,2,2}, \beta_{n,k,3,1}, \beta_{n,k,4,0}\}$. Note that in the code, the alpha coefficients, for ease of use in this specific case, are represented by a 2-dimensional array.

```
//r=2 , m=2

double[] beta_array = new double[15];
int i = 0;
for (int k = 0; k <= 2 * r; k++)
 for (int r1 = 0; r1 <= k; r1++) {
  int r2 = k - r1;

  // compute gamma for r1 and r2

  double gamma =
   computeGamma(alpha, r, r1, r2);
  if (k <= 2) {
   beta_array[i++] =
    alpha[r1][r2] * z - gamma;
  } else {
   beta_array[i++] = -1 * gamma;
  }
 }
return beta_array;
```

where *computeGamma(alpha, r, r1, r2)* is

```
double gamma = 0;

// r1' = a, r2'= b, r1" = c, r2"= d

for (int a = 0; a <= r; a++) {
int c = r1 - a;
if (c >= 0 && c <= r)
 for (int b = 0; b <= r; b++) {
  int d = r2 - b;
  if (d >= 0 && d <= r)
   gamma +=
    alpha[a][b] * alpha[c][d];
 }
}
return 0.5 * gamma;
```

The sensor is now ready to begin the consensus algorithm we discussed in section 2.4.2. Note that along with the $\tilde{\beta}_{n,k,r_1,r_2}$ coefficients, the normalization factor $\tilde{C}_n(\mathbf{z}_n)$ is also computed via consensus on the logarithm (2.21).

1. For the coefficient vector and the normalization factor, the old internal state is retrieved or created from scratch in the case the sensor is at its first PF iteration.

2. A temporary state for the vector and the normalization factor is updated from the old internal state, using current information

```
//tp is the tuning parameter
computeTempState(double[] current,
  double[] prev_is, double tp) {
  double[] temp_is =
   new double[current.length];
  for (int i = 0; i < temp_is.length; i++)
   temp_is[i] =
     tp * prev_is[i] + (1 - tp) * current[i];
  return temp_is;
  }
```

3. The updated temporary state is broadcast to all neighboring sensors with the approriate *send* method offered by the Jason library (the list of neighbors is previously retrieved from the belief base)

4. Disregarding synchronization, the currently available temporary states broadcast by other sensors are retrieved from the agent's belief base and, in a single iteration, the Metropolis weight is calculated (cfr. 2.26) and used in the internal state computation (including the normalization factor). After the cycle has ended, the local temporary state is added to the new internal state, in order to facilitate the computation of the Metropolis weight for the local state.

```
//acquire belief from agent's belief base

belief = agent.getBB().getCandidateBeliefs(

//supply the format of the belief needed

Format.literal("temp_state(_,_)[trackable(%s)]",
   trackable),
un);

//if some belief is found

if (belief != null)

//cycle through all beliefs with that format

while (belief.hasNext()) {
  Literal lit = belief.next();

//obtain the temporary coefficients

  double[] state =
  Format.numberListToArray((ListTerm) lit.getTerm(0));

//obtain the temporary normalization factor

  double c =
```

```
        ((NumberTerm) lit.getTerm(1)).solve();

    //compute Metropolis weights
    //the number of neighbors a sensor has is carried along
       with the
    //broadcast temporary state as an annotation

      double weight = 1 / (1 + Math.max((double)
         neighbors.size(),
       ((NumberTerm)
        lit.getAnnot("neighbors")
             .getTerm(0)).solve()));
      totalw += weight;

    //the weighted temporal state is added to the new
       internal state

      for (int j = 0; j < internal_state.length; j++)
       internal_state[j] += weight * state[j];
      norm_const += weight * c;
    }

    //local temporary state is added

    double weight = 1 - totalw;
    for (int j = 0; j < internal_state.length; j++)
            internal_state[j] += weight * temporary_state[j];
    norm_const += weight * temp_norm_const;

    [...]
    //all the internal state elements are multiplied
    //by the number of sensors in the whole network

    for (int j = 0; j < internal_state.length; j++)
            likelihood_parameters[j] = internal_state[j] * k;
    norm_const = k * norm_const;
```

Finally, a JLF object can be instantiated with the coefficients and the normalization C_n factor as parameters (Note that the normalization factor at this point is actually its logarithm, so before being multiplied to the exponential it needs to be reverted). When the method *weight(particle)* is called, the approximated exponential sum (2.42) is computed, and the complete approximation of the JLF (cfr. 2.12) is evaluated for the particle given as input.

```
public double weight(double[] state) {
 double sum = 0;
 int i = 0;
 for (int k = 0; k <= 4; k++)
  for (int r1 = 0; r1 <= k; r1++) {
   int r2 = k - r1;
    sum += coefficient[i++]
     * Math.pow(state[0], r1)
```

```
      * Math.pow(state[1], r2);
  }
 return Math.exp(sum + logcn);
}
```

4.3 Possible improvements

Now that we have a complete insight on the whole implementation and the engineering difficulties that we encountered, we took a moment to rehearse what we've and think to what could be improved. Certainly, in this primordial stage, the application might not yet be suitable for research or other kinds of application since it lacks some quality-of-life improvements that would make this software much more versatile and desirable.

Some of the improvement ideas are listed as follows.

- For now the application works only with a very specific set of parameters. A big improvement would be editing the each procedure in the particle filtering algorithm so that it can accept a variable set of parameters, so that different experiments can be performed.
- Currently the application only works for the tracking of a mobile object position in a 2-dimensional virtual space, though distributed particle filtering can be applied for any kind of state vector, provided there is a set of sensors capable of measuring all the dimensions present in the state vector. On this idea another huge improvement would be having different measurement functions which can be set as parameters for the sensor. The logic of the sensor would be the same, but the kind of measurement changes. In this way a heterogeneous sensor network could be simulated and tested.
- Having a set of different sensor logic, each for any kind of distributed particle filtering configuration. This way the user could quickly set up a new network and test how different configurations perform.
- Small-grain modularity, letting the user select different functions to compose the algorithm e.g. choosing among different resampling methods.
- A wider range of testing and data collection tools should made available to the user, such as simulated transferred data volume counters in the case a wireless network needs to be tested, or simply a wider range of statistical tools measuring accuracy and other kinds of performance indicators.
- Overall a better looking and functional GUI.

Chapter 5

Conclusions

We began this project with the idea of having an insight on what the mathematical notions and the technical difficulties are when a Distributed Multi-sensing Data Fusion System needs to be implemented. Along the way we found out that Particle filtering is one tool, among others, that best performs when it's time to fuse numerical -data perceived by different sensors- into a dynamic world model. This led us to closely explore this data fusion practice in literature, and try to use it in the implementation of a distributed agent system software so to better understand the process and face the aforementioned technical difficulties first-hand. For this we defined a specific case study of target tracking onto where we applied the acquired knowledge for distributed particle filtering. The case study brought us to implement a small target tracking simulator software application using Java, with the Jason library, and AgentSpeak languages. The outcome was an application that allowed us to try out this novel technique that addresses a desirable feature in located autonomous systems, that is a reliable process able to dynamically model the environment in which these systems are immersed. The application that has been developed only covered a very specific case scenario, but allowed us to grasp the core mechanisms of likelihood consensus distributed particle filtering. Consequently this enables us to more easily apply those same mechanisms to many different and more complex scenarios, by only changing the architectural requirements.

Bibliography

[1] URL: http://jason.sourceforge.net/wp/.

[2] URL: http://commons.apache.org/proper/commons-math/.

[3] M. Sanjeev Arulampalam et al. "A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesan Tracking". In: *IEEE Transactions on Signal Processing* 50.2 (Feb. 2002), pp. 174–188.

[4] Rafael H. Bordini, Jomi Fred Hubner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*. Wiley, 2007.

[5] James L. Crowley and Yves Demazeau. *Principles and Techniques for Sensor Data Fusion*. LIFIA (IMAG), 46 avenue Félix Viallet, F-38031 Grenoble Cédex, France.

[6] Ondrej Hlinka, Franz Hlawatsch, and Petar M. Djuric. "Distributed Particle Filtering in Agent Networks". In: *IEEE Signal Processing Magazine* (Nov. 2013), pp. 61–81.

[7] Ondrej Hlinka et al. "Likelihood Consensus ad its application to Distributed Particle Filtering". In: *IEEE Transactions on Signal Processing* 60 (8 2012), pp. 4334–4349.

[8] Ondrej Slučiak et al. "Sequential likelihood consensus and its application to distributed particle filtering with reduced communications and latency". In: *2011 Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)* (2011), pp. 1766–1770.